Microsoft .Net Framework and C# Programming Language

Lesson 7

Exception handling. Garbage collector

Contents

1. **Exception hierarchy 4**
   - The System.Exception base class 5
   - The analysis of standard exception hierarchy. 6
   - Inheritance and exceptions. 7
   - Inheritance from standard exceptions classes 8
2. **The basics of exception handling 11**
   - The try keyword 11
   - The catch keyword 13
   - The finally keyword 14
   - The throw keyword 17
3. **The ins and outs of exception handling 20**
   - Catching all the exceptions 23
   - The try nested blocks 28
   - Exception regeneration 31

4. Applying the checked and unchecked keywords 37
5. Exception filters 42
6. The use of nameof 45
7. Life cycle of the objects 47
8. The concept of garbage collector 48
9. The concept of generations in garbage collection 50
10. The System.GC class 52
11. Finalizer and the Finalize method. 55
12. The Dispose method and IDisposable interface 59
13. The use of using when working with classes that implement IDisposable interface. 63
14. Home task 69

1. Exception hierarchy

The code that does not contain errors is an ideal that is rarely found in practice, but it is still worth aspiring to. There is even a special area in the theory of reliability, which is applied for theoretical estimate of the number of errors in a project, depending on the number of modules, development technology used, and other factors as well.

In sober fact, the reason for failures at application execution is not always the result of errors in a program code. The errors can be caused by incorrect actions of a user, or some external factors, such as hardware failures, or unavailability of some resources (for example, a network drive, or a database server).

However, this does not in any way reduces the responsibility of a programmer who developed this software. Therefore, a professionally designed application should provide the correct handling of any errors occurred.

Many different approaches for error handling were used before the advent of the .Net Framework technology. For example, many functions of WinAPI return an error indication (for example, FALSE, INVALID_HANDLE_VALUE, NULL) at unsuccessful execution. Then, using the `GetLastError()` function, you can get use an error code to find its description. This way of error handling is time-consuming, since it is necessary to check the return result after calling a function, and it is also not quite reliable, because it is possible to proceed without checking the result as if the function has been executed successfully.

The so-called structured exception handling is used in the .Net Framework, which generates and detects errors in the runtime environment. The advantages of this technique

lie in homogeneous error handling regardless of the language of the .Net family. Another advantage of this approach is that the exceptions are handled in the same way regardless of the technology used, whether it is a service, desktop or web application. In addition, any exception contains a set of properties that allow getting detailed information about it. It is possible, because each exception is represented by the heir class of the `System.Exception` base class or its descendant.

The System.Exception base class

In C#, the base class for all exceptions is the `Exception` class that contains a number of virtual methods and read-only properties that contain additional information about an error, which makes it easier to debug a program (Table 1.1).

Table 1.1. System.Exception class properties

Property name	Description
`string Message`	Contains the message text indicating the cause of exception.
`IDictionary Data`	A reference to a "parameter-value" set of pairs. Usually, the code that generates an exception add entries to this set. The code that caught an exception can use this data for more information about the cause of exception.
`string Source`	Contains the name of an assembly that thrown an exception.
`string StackTrace`	Contains a sequence of methods, the call of which caused an exception.
`MethodBase TargetSite`	Contains the MethodBase object describing a method that threw an exception.

Property name	Description
`string HelpLink`	Contains the URL of a document describing an exception
`Exception InnerException`	Contains the previous exception that caused the current exception

Exception is an object that is created and "thrown" in the case of an error. Any CLS-compliant language should be able to throw and catch the types of exceptions derived from the `System.Exception` base class. These exceptions are called CLS-compliant.

The analysis of standard exception hierarchy

The BCL has many exception classes that can be used by a developer. He can also create his own exception classes. Figure 1.1 partially shows the exception classes diagram.

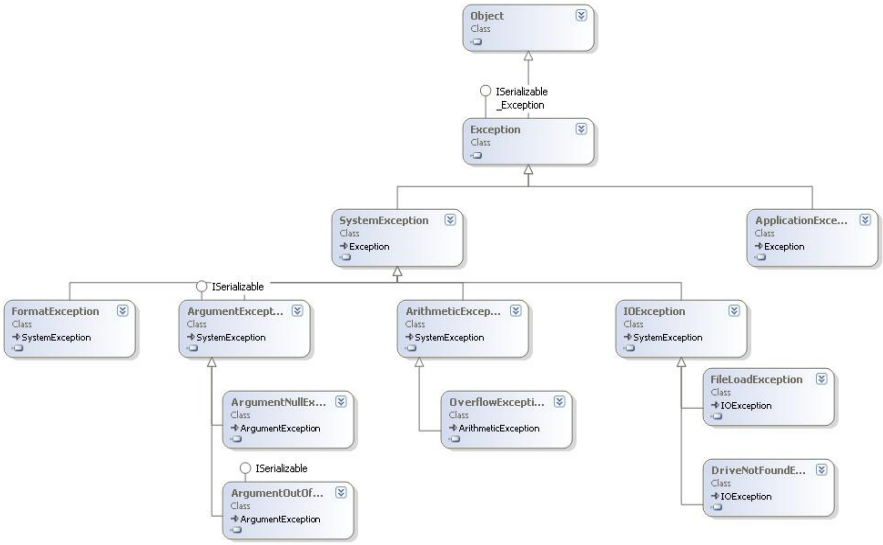

Figure 1.1. Exception class hierarchy (fragment)

It should be emphasized that the Figure 1.1. shows not all of the exceptions. The main purpose of this diagram is to show the general laws of exception class hierarchy.

You've probably noticed that only two classes are directly inherited from the base Exception class: `SystemException` and `ApplicationException`. They are the base for the other classes in the exception inheritance hierarchy.

Inheritance and exceptions

The `SystemException` class is a base class for the exceptions that are typically thrown by the CLR, or are the exceptions of general nature and can be thrown by any application. For example, the `StackOverflowException` exception is thrown by the CLR if there is a stack overflow, while the `ArgumentException` exception (and its derivatives) can be thrown by any application if the method receives invalid argument values.

All the specific exceptions created by the developers should be inherited from the `ApplicationException` class.

Exception class inheritance is somewhat unusual, as the derived classes do not generally add new functionality to the features of a base class, but are used to indicate more specific cause of an error. For example, the `ArgumentNullException` (thrown when passing null value as a parameter to a method that does not accept it as valid) and `ArgumentOutOfRangeException` (thrown when an argument violates the allowable range of values) classes are inherited from the `ArgumentException` class.

Inheritance from standard exceptions classes

When creating a special exception in his programs, the developer must create an heir class from the `ApplicationException` class, which is a recommendation aimed at determining the cause of an error. When handling such an exception, it will be clear that an error is caused by incorrect operation of the classes of an application itself, rather than the `.Net Framework` base classes.

There are three ways to create special exceptions. Let us consider each of them on the example of an exception, the properties of which will contain the time and some information.

Since the exceptions are the classes, then the first way lies in creating a class with members required to work with it. To do this, we redefine the property of the Message base class.

```
public class MyException : ApplicationException
{
    private string _message;
    public DateTime TimeException { get; private set; }

    public MyException()
    {
        _message = "My exception";
        TimeException = DateTime.Now;
    }

    public override string Message
    {
        get
        {
            return _message;
        }
    }
}
```

Actually, there is no special need in redefinition of base class properties, because we can call the base class constructor to transfer our information. This is the second way to create a special exception.

```
public class MyException : ApplicationException
{

    public DateTime TimeException { get; private set; }

    public MyException() : base("My exception")
    {
        TimeException = DateTime.Now;
    }
}
```

The third way is to implement the recommendations of the .Net Framework on creation of a special exception class, which are as follows:

- exception should be the heir of `ApplicationException`;
- the [`Serializable`] attribute should be specified for an exception (this will be discussed in later lessons);
- exception should have a default constructor;
- exception should have an overloaded constructor to set the `Message` property;
- there should be an overloaded constructor for internal exception handling;
- there should be an overloaded constructor that performs type serialization.

There is no more need to write a lot of code to fulfill all of these recommendations; it is enough to use a special feature

of the Visual Studio. Just write the Exception keyword and press the Tab key twice (Figure 1.2).

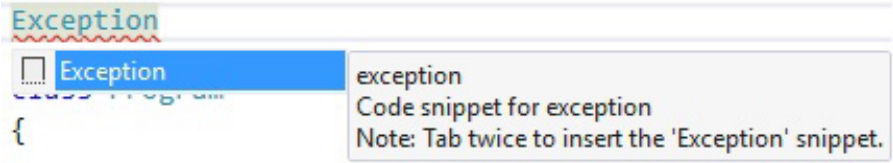

Figure 1.2. Creating a custom exception class

Visual Studio will generate a class, and we will only need to make the necessary changes, then we'll get the same functionality as in the first two cases.

```
[Serializable]
public class MyException : Exception
{
    public DateTime TimeException { get; private set; }
    public MyException() : this(«Мое исключение»)
                        { TimeException = DateTime.Now; }
    public MyException(string message) : base(message) { }
    public MyException(string message, Exception inner)
                        : base(message, inner) { }
    protected MyException(System.Runtime.Serialization.
                        SerializationInfo info,
                        System.Runtime.Serialization.
                        StreamingContext context)
                        : base(info, context) { }
}
```

It is worth noting that the resulting class inherits the `Exception` class, not the `ApplicationException` class as we expected. What a mystery…

2. The basics of exception handling

A special block is applied to handle any type of exceptions: try-catch-finally. Here is the syntax:

```
try
{
//the code, in which an exception may occur
}

catch(exception _ type)
{
//exception handling
}

finally
{
//resource deallocation
}
```

As you can see, this block consists of three separate blocks, each of which is created with the same keywords. Let us describe them.

The try keyword

The `try` block is designed for placing statements in it, the implementation of which can potentially cause an exception.

The `try` block must be used only in conjunction with the `catch` or `finally` block, or both. The use of the `try` block alone is prohibited. The following code will cause an error in Figure 2.1.

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1, number2, result = 0;

            WriteLine("Enter two numbers");

            try
            {
              number1 = int.Parse(ReadLine());
              number2 = int.Parse(ReadLine());

              result = number1 / number2;

              WriteLine($"The quotient: { result }");
            }
        }
    }
}
```

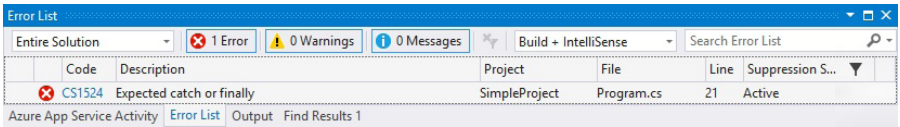

Figure 2.1. Error: catch or finally is expected

The process of creating exception handling blocks is also simplified thanks to special features of Visual Studio. You just need to write the word `try`, and a list of options will be displayed (Figure 2.2).

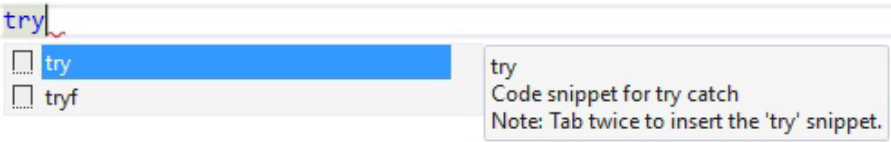

Figure 2.2. Creating exception handling block

Press the tab key after selecting one of the items you need. If try was selected from the list, then a template of the `try-catch` block will be generated, and the selection of tryf will generate a `try-finally` block.

The catch keyword

The `catch` block contains the code that will be executed only when an exception occurs. When declaring the `catch` block, the type of exception to be handled is usually specified. If the `try` block is executed without an exception, then the `catch` block will not be executed (Figure 2.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1, number2, result = 0;
            WriteLine("Enter two numbers");
            try
            {
              number1 = int.Parse(ReadLine());
              number2 = int.Parse(ReadLine());
              result = number1 / number2;
```

```
                WriteLine($"Quotient: { result }");
            }
            catch (DivideByZeroException de)
            {
                WriteLine(de.Message);
            }
        }
    }
}
```

Possible outcome:

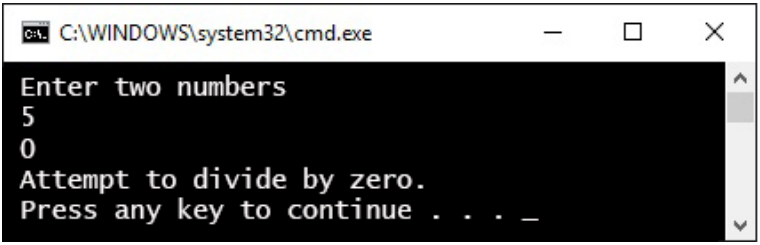

Figure 2.3. Handling a forbidden operation

The finally keyword

The `finally` block usually contains actions that must be performed regardless of whether there was an exception in the `try` block, or not. In other words, the `finally` block is always executed regardless of an exception occurrence (Figure 2.4).

```
using System;

using static System.Console;
namespace SimpleProject
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            WriteLine("Enter two numbers");
            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
                result = number1 / number2;

                WriteLine($"The quotient is {result}");
            }

            catch (DivideByZeroException de)
            {
                WriteLine(de.Message);
            }

            finally
            {
                WriteLine("Resource deallocation");
            }
        }
    }
}
```

Possible outcome:

```
C:\WINDOWS\system32\cmd.exe
Enter two numbers
48
6
The quotient is 8
Resource deallocation
Press any key to continue . . . _
```

Figure 2.4. Example of the finally block execution

The `finally` block may contain code for some resource deallocation, such as closing a file or a connecting to a database. If there is no need to perform any resource deallocation, the `finally` block may be missing.

It should be noted that the use of the `try` and `finally` blocks without the `catch` block is syntactically permissible, but there will be no exception handling, and in case of its occurrence the program will crash (Figure 2.5).

```
using static System.Console;
namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Enter two numbers");
            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
                result = number1 / number2;
                WriteLine($"The quotient is {result}");
            }

            finally
            {
                WriteLine("Resource deallocation");
            }
        }
    }
}
```

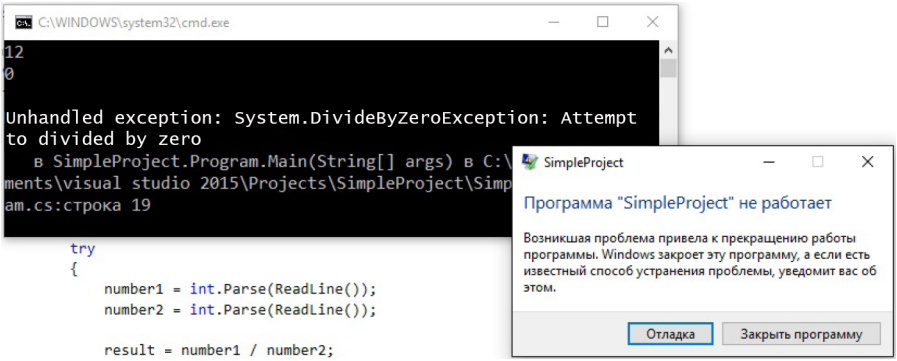

Figure 2.5. Runtime error

The throw keyword

The `throw` keyword is used to explicitly generate an exception at runtime. The syntax of explicit throwing looks the following way:

```
throw new Exception _ type();
```

A class inherited from the `Exception` class should be used as an exception type, which describes the problem in the best way. It is better not to use the `SystemException` or `ApplicationException` base classes, as it will be difficult to define the cause of an exception when handling it.

Here is an example of explicit exception generation, with the special exception called `MyException`, the examples of which creation were discussed by us in Section 1.

```
using System;
using static System.Console;

namespace SimpleProject
{
```

```
    public class MyException : ApplicationException
    {
        public DateTime TimeException { get; private set; }
        public MyException() : base(("My exception")
        {
            TimeException = DateTime.Now;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            WriteLine(("Enter two numbers");
            int number1, number2, result = 0;
            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
               result = number1 / number2;
                if (result % 2 != 0)
                {
                    throw new MyException();
                }

                WriteLine($"Quotient: { result }");
            }
            catch (MyException my)
            {
                WriteLine(my.Message);
                WriteLine(my.TimeException);
            }
        }
    }
}
```

Possible outcome (Figure 2.6).

```
C:\WINDOWS\system32\cmd.exe
Enter two numbers
75
8
My exception
12.10.2016 10:26:24
Press any key to continue . . . _
```

Figure 2.6. Explicit generation of special exception

3. The ins and outs of exception handling

The exceptions are handled in the following way: when an exception occurs in the `try` block, its execution is terminated, and the CLR searches the `catch` block intended to handle this type of exception. If this block is found, it is executed, then the `finally` block (if any) is executed. If the appropriate `catch` block has not been found, then the exception is passed up the call stack in search of the required `catch` block in the previous methods. The last in this chain is the `Main()` method, and if the exception is not handled in it, then the exception is caught by the CLR, which crashes the program (Figure 2.5).

The sequence of execution of the program when an exception occurs is demonstrated by the following code (Figure 3.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                WriteLine("The code before exception");//1
                throw new Exception("Test Exception"); //2
```

```
                //This line will never appear
                WriteLine("The code after exception");
            }

            catch (Exception e)
            {
                WriteLine($"Error: { e.Message }"); //3
            }

            finally
            {
                WriteLine("The code of the finally block");//4
            }
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
The code before execution
Error: Test Exception
The code of the finally block
Press any key to continue . . . _
```

Figure 3.1. The execution sequence of the try-catch-finally block statements

The text "The code after exception" will not be displayed, because execution of the `try` block is terminated after throwing an exception.

It is also necessary to understand that your program will continue its execution after an exception has been successfully handled. And you should be aware that the use of uninitialized variable outside the `try-catch` block will crash your program (Figure 3.2).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Enter two numbers");
            int number1, number2, result = 0;

            string str = null;


            try
            {
               number1 = int.Parse(ReadLine());
               number2 = int.Parse(ReadLine());
               result = number1 / number2;
               str = "Check";
               WriteLine("Quotient: " + result);
            }

            catch (DivideByZeroException de)
            {
                WriteLine(de.Message);
            }

            WriteLine(str.Contains("successful"));

        }
    }
}
```

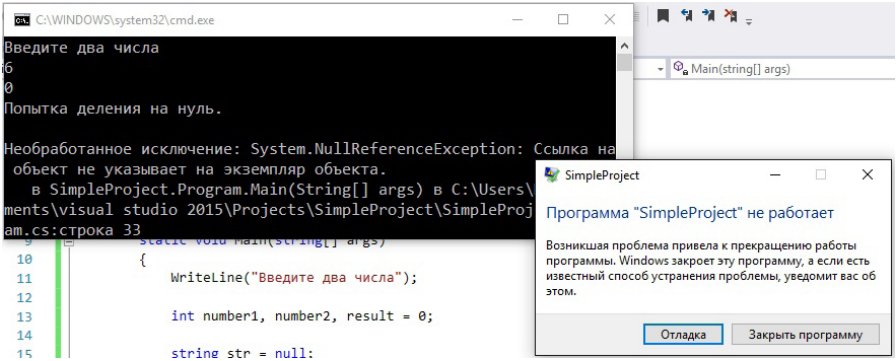

Figure 3.2. Unhandled exception after the try-catch block

Catching all the exceptions

Since the `System.Exception` is the base class for all exception classes, then you can use a catch block that `catches` this particular class to handle all types of CLS compliant exceptions:

```
catch (Exception e)
{
    WriteLine(e.Message);
}
```

As already noted, the C# allows generating only those exception classes derived from `System.Exception`. However, the methods developed in other programming languages, such as C++, can generate any type of exception. To handle all types of exceptions (including not CLS-compatible) in the `catch` block, you can not to specify the type of exception to be caught:

```
catch
{
    WriteLine(«Error!!!»);
}
```

However, in this case there is no possibility to obtain detailed information about the error, and you can specify only a universal information.

There are often situations when the `try` block contains the code, the execution of which could lead to different types of errors. In such cases, it is necessary to use several `catch` blocks, each of which will handle its error type, if necessary. As mentioned above, the CLR searches for an appropriate `catch` block to handle the error by iterating through all the blocks from top downward. Therefore, they must be placed from more individual cases to more general ones, and the last of them should be the handling block of the `System.Exception` base class. Improper placement of `catch` blocks will lead to an error at compile time (Figure 3.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Enter two numbers");
           int number1, number2, result = 0;
            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
                result = number1 / number2;
                WriteLine("Quotient: " + result);
            }
```

```
            catch (Exception e)
            {
                WriteLine(e.Message);
            }
            catch (DivideByZeroException de)
            {
                WriteLine(de.Message);
            }
        }
    }
}
```

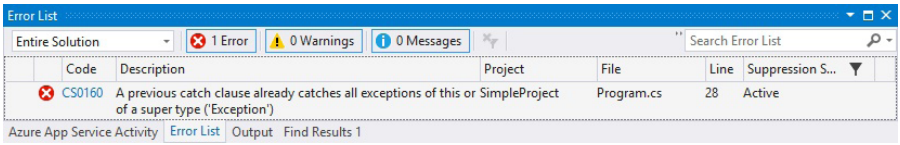

Figure 3.3. Error: The block that precedes the catch block already catches all the exceptions

To demonstrate the use of multiple `catch` blocks, let us consider the following example: calculating the expression 100/ln(n), where n is a number entered by the user. The following types of exceptions will be caught in the `catch` blocks:

- `FormatException` occurs if the line entered by the user cannot be converted to a number;
- `DivideByZeroException` occurs if ln(n) = 0, i.e. n = 1;
- `Exception` includes all other exceptions.

```
using System;
using static System.Console;

namespace SimpleProject
{
```

```
class Program
{
    static void Main(string[] args)
    {
        do
        {
            try
            {
                Write($"\nEnter a number: ");
                //reading user input
                string str = ReadLine();
                //loop exit condition

                if (string.IsNullOrEmpty(str))
                {
                    return;
                }
                //converting line to a number
                int number = Convert.ToInt32(str);
                //checking that the received number
                //belongs to the interval on which
                //the function ln() is defined
                if (number <= 0)
                {
                    throw new
                      ArgumentOutOfRangeException
                   ("number  <= 0");
                }
                double log = Math.Log(number);
                WriteLine($"ln({number}) =
                         {log}\n100/ln({number}) =
                         {100 / (int)log}");
            }

            catch (FormatException fe)
            {
```

```
                    //occurs if the number entered
                    //by the user cannot be converted
                    //to an integer
                    WriteLine(fe.Message);
                }

                catch (DivideByZeroException de)
                {
                    //occurs if Log(n) = 0 (i.e. n = 1)
                    WriteLine(de.Message);
                }

                catch (Exception e)
                {
                    //catching all other exceptions
                    WriteLine($"Exception: { e.Message }");
                }

            } while (true);
        }
    }
}
```

Program outcome (Figure 3.4).

```
C:\WINDOWS\system32\cmd.exe
Enter a number: 0
Exception: Argument is out of range.

Parameter name: number <= 0

Enter a number: 1
Attempt to divide by zero.

Enter a number: 5
ln(5) = 1,6094379124341
100/ln(5) = 100
```

Figure 3.4. The use of multiple catch blocks

The try nested blocks

The `try` blocks can be nested; their general form is shown below:

```
try                     //external block
{
    //point A
    try                 //internal block
    {
        //point B
    }

    catch
    {
        //point C
    }
    finally
    {
        //deallocation
    }
    //point D
}
catch
{
    //exception handling
}
finally
{
    //deallocation
}
```

Consider the possible scenarios for generation and handling of an exception in this block. If an exception occurs at point B, then the program searches the appropriate exception handler. If an exception can be handled by the internal `catch` block,

it is caught and handled, then the code execution continues (point D). If there is no new exception at the point D, the `catch` block of the external block is ignored.

If an exception cannot be caught by the `catch` or an error occurred in point D, then the external catch block is checked. If this block cannot handle the exception, an appropriate handler is searched higher in the call stack.

The `finally` blocks of the nested and external blocks are executed in any case.

Consider the example of the nested `try` blocks. Two types of exceptions occur in the internal block: division by 0, and addressing an array by an invalid index. The first exception is caught by the internal `catch` block, while the second exception is caught by the external block (Figure 3.5).

```
using System;
using static System.Console;
namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] a = new int[5];
            int n = 0;
            try //external try block
            {
                for (int i = -3; i <= 3; i++)
                {
                    //the loop termination does not occur
                    //when dividing by 0: the exception
                    //is caught and the exception is caught
                    try //and internal try block
                    {
                        a[n] = 100 / i;
                        WriteLine(a[n]);
                        n++;
                    }
                    catch (DivideByZeroException e)
                    {
                        WriteLine("Internal catch");
                        WriteLine(e.Message);
                    }
                }
            }
            catch (IndexOutOfRangeException e)
            {
                WriteLine("External catch");
                WriteLine(e.Message);
            }
        }
    }
}
```

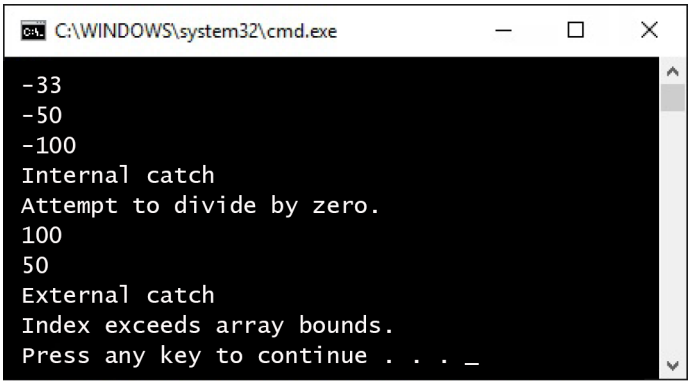

Figure 3.5. Example of the try nested blocks

Exception regeneration

Exception regeneration is performed only in the `catch` block with the use of the `throw` statement.

In the first case, the `throw` statement is used without specifying the exception type. The following syntax can be used, when it is necessary to forward an exception to the caller code from the `catch` block (Figure 3.6).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static int DivisionNumbers(int n1, int n2)
        {
            int result = 0;

            try
            {
```

```
                result = n1 / n2;
            }

            catch (DivideByZeroException)
            {
                throw; //forwarding an exception up
                       //the call stack
            }
            return result;
        }

        static void Main(string[] args)
        {
            WriteLine("Enter two number");

            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());

                result = DivisionNumbers(number1, number2);

                WriteLine($"Quotient: { result }");
            }

            catch (Exception e)
            {
                WriteLine(e.Message);
            }
        }
    }
}
```

Enter two numbers

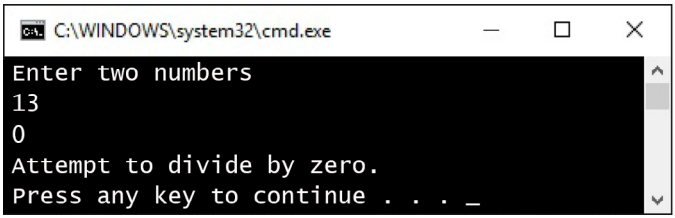

Figure 3.6. Exception regeneration

In fact, when regenerating an exception, it is recommended to forward additional information as well, so the debug process would become more informative. The usual exception throwing syntax is used to generate an exception of this type. Original exception caught by the catch block is stored in the InnerException field of a new exception (Figure 3.7).

```
using System;
using static System.Console;
namespace SimpleProject
{
    class Program
    {
        static int DivisionNumbers(int n1, int n2)
        {
            int result = 0;
            try
            {
                result = n1 / n2;
            }
            catch (DivideByZeroException de)
            {
                throw new Exception("Additional
                                    information", de);
            }
            return result;
        }
```

```
        static void Main(string[] args)
        {
            WriteLine("Enter two numbers");
            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
                result = DivisionNumbers(number1, number2);
                WriteLine($"Quotient: { result }");
            }

            catch (Exception e)
            {
                WriteLine(e.Message);
                //additional information

                WriteLine(e.InnerException.Message);
                //previous exception
            }
        }
    }
}
```

Possible outcome.

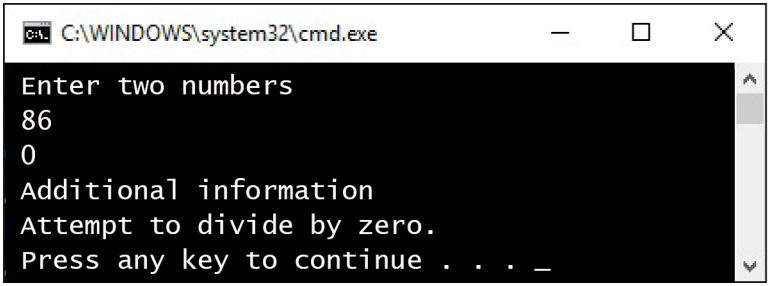

Figure 3.7. Exception regeneration with additional information

When throwing an exception, the CLR registers a method, in which the `throw` statement has been executed, or an exception has been thrown. When detecting the `catch` block that can handle this exception, the CLR registers the location where this exception was caught. Now, if we address the StackTrace property of a generated exception object in the `catch` block, the code of this property will call the CLR. This CLR will build a line that identifies all of the methods called between the occurrence of an exception and its handling. The StackTrace property provides information about exception handling, including the method that generated it.

An example of stack trace at exception occurrence is shown below. In this case, the exception occurs in the f2() method, and is caught in the f() method (Figure 3.8).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void f2()
        {
            throw new Exception("MyException");
            //throwing an exception
        }
        static void f1()
        {
            f2();
        }
        static void f()
        {
```

```
            try
            {
                f1();
            }
            catch (Exception e)
            //catching an exception
            {
                WriteLine(e.Message);
                //writing a stack trace
                WriteLine("Stack trace:\n {0} ", e.StackTrace);
            }
        }
        static void Main(string[] args)
        {
            f();
        }
    }
}
```

Possible outcome.

```
C:\WINDOWS\system32\cmd.exe
MyException
Stack trace:
    в SimpleProject.Program.f2() в C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProjec
t\Program.cs:строка 11
   в SimpleProject.Program.f1() в C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject
\Program.cs:строка 15
   в SimpleProject.Program.f() в C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject\
Program.cs:строка 21
Press any key to continue . . . _
```

Figure 3.8. Example of stack trace

The f(), f1(), f2() methods are written in a trace, while the Main() method is not written, since it is located higher in the call stack, than the f() that caught an exception.

4. Applying the checked and unchecked keywords

An overflow may occur when performing arithmetic operations on integer data types. This is a situation, in which the number of bits of the result exceeds the capacity of the variable, to which this result is written.

An overflow may occur in the following situations:

- In expressions that use arithmetic operators;
- when explicitly casting integer types.

The CLR can use one of two options when an overflow occurs:

- ignoring an overflow and discarding digits of higher order;
- generating an `OverflowException`.

By default, the bits of higher order are discarded when an overflow occurs (Figure 4.1).

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            byte b = 100;
            b = (byte)(b + 200);
            //digits that do not fall within the range
            //of the Byte are discarded
            WriteLine($"(byte)300 = {b}"); // 44
            int n = 65536;
            short s = (short)n;
            //digits that do not fall within the range
            //of the Short are discarded
            WriteLine($"(short)65536 = {s}"); // 0
        }
    }
}
```

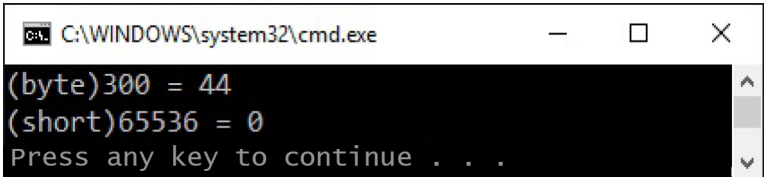

Figure 4.1. An overflow at explicit cast

Overflow checking mode can be specified explicitly using the `checked` and `unchecked` keywords.

The `checked` keyword sets an overflow checking mode with the exception generation.

The `unchecked` keyword sets the ignoring of overflow occurrence.

The following example demonstrates how to use these keywords. In this case, the value with an initial value of 255 will be increased by one: an exception occurs in the `checked` block, and is caught in the `catch` block, while the `unchecked` block does not throw an exception, but just writes 0 to a variable (Figure 4.2).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            byte b = 255;
            try
            {
                checked
                {
                    b++; //generating OverflowException
                }
                WriteLine(b);
            }
            catch (OverflowException e)
            {
                WriteLine(e.Message);
            }
            try
            {
                unchecked
                {
                    b++; //overflow is ignored
                }
                WriteLine(b); //0
            }
            catch (OverflowException e)
            {
                WriteLine(e.Message);
            }
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
Переполнение в результате выполнения арифметической операции.
0
Для продолжения нажмите любую клавишу . . .
```

Figure 4.2. The use of checked and unchecked keywords

The checked and unchecked keywords can be used as statements in a conversion line. The following example demonstrates explicit cast of an integer to the Byte type (Figure 4.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            byte b = 100;
            WriteLine(unchecked((byte)(b + 200))); // b = 44

            try
            {
                WriteLine(checked((byte)(b + 200)));
                //throwing an exception
            }
            catch (OverflowException oe)
            {
                WriteLine(oe.Message);
            }
        }
    }
}
```

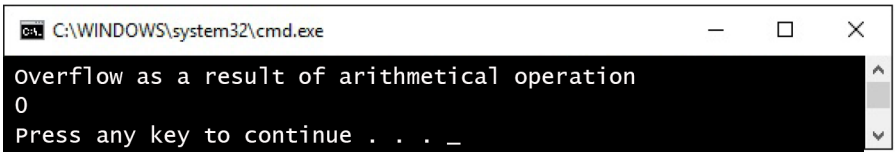

Figure 4.3. The use of checked and unchecked keywords in a line

If the overflow checking mode was not explicitly specified, then it is defined by the /checked compiler option. This option can be set using the Properties item of the Project menu. To do this, press the Advanced… button in the Build tab, and then check or uncheck the Check for arithmetic overflow/underflow box in the opened form. If checked, the exception (Figure 4.4) is generated when an overflow occurs.

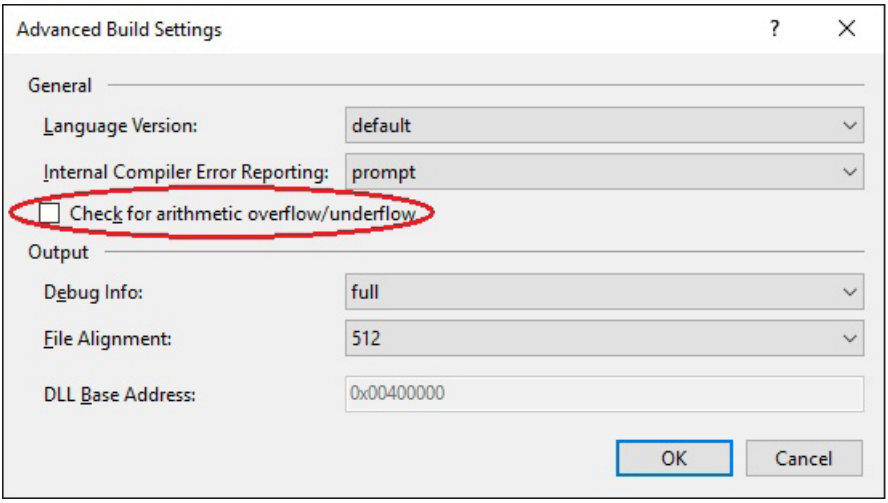

Figure 4.4. Advanced Build Settings

5. Exception filters

In C# 6.0 the support of exceptions was extended by the introduction of such feature, as exception filters. These filters made possible to handle exceptions on the basis of additional conditions specified in the `catch` block. General form looks the following way:

```
try
{
//the code, in which an exception may occur
}

catch(exception_type) when (conditon)
{
//exception handling
}
```

Here is an example of an exception filter, it which the first `catch` block will be executed only if the current exception was caused by another one, while the second `catch` block will be executed in any other cases (Figure 5.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static int DivisionNumbers(int n1, int n2)
        {
```

```
        int result = 0;

        try
        {
            result = n1 / n2;
        }

        catch (DivideByZeroException de)
        {
            throw new Exception("Checking exception
                                 filters", de);
        }
        return result;
    }

    static void Main(string[] args)
    {
        WriteLine("Enter two numbers");
        int number1, number2, result = 0;

        try
        {
            number1 = int.Parse(ReadLine());
            number2 = int.Parse(ReadLine());
            result = DivisionNumbers(number1, number2);

            WriteLine($"Quotient: { result }");
        }

        catch (Exception e) when (e.InnerException !=
                                  null)
        {
            WriteLine(e.Message); //additional
                                  //information
            WriteLine(e.InnerException.Message);
            //previous exception
        }
```

```
            catch (Exception e)
            {
                WriteLine(e.Message);
            }
        }
    }
}
```

Possible outcome.

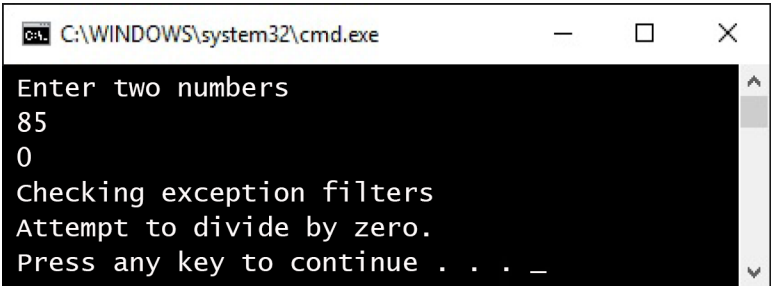

Figure 5.1. The use of exception filter

6. The use of nameof

Another improvement of C# 6.0 is the `nameof` statement that allows getting the name of a class or property in the form of a string.

Before that, the developer had to use the so-called "magic strings" in the code, particularly, when throwing the `ArgumentException` exception (and its heirs). In some cases, it is necessary to forward the name of a parameter that caused an exception (a similar case was shown in Section 3). Unfortunately, when writing a "magic string", there is a high probability to commit a syntax error. Another important reason is that the renaming of an element will not automatically update its string representation, which will lead to inconsistency that will manifest itself only at runtime.

The application of the `nameof` statement allows avoiding syntax errors in cases, where it is necessary to specify a string literal. Also, the use of the `nameof` statement helps to keep the code correct when renaming elements, which is very important in case of further refactoring. An example of the `nameof` statement is shown in Figure 6.1.

```
using System;
using static System.Console;

namespace SimpleProject
{
    class ExampleNameOf
    {
        public string Name { get; set; }
        public ExampleNameOf(string name)
        {
```

```
            if (name == null)
            {
               throw new ArgumentNullException(nameof(name));
            }

            Name = name;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            try
            {
               ExampleNameOf example = new ExampleNameOf(null);
            }
            catch (Exception e)
            {
                WriteLine(e.Message);
            }
        }
    }
}
```

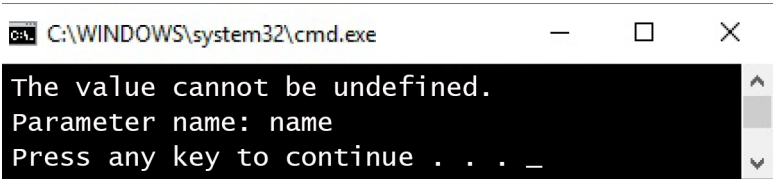

Figure 6.1. Example use of the nameof statement

7. Life cycle of the objects

Virtually any program uses a variety of objects, the work with which leads to the consumption of RAM. In object-oriented programming, each type describes an object, with which the program will work after its creation.

The life cycle of any object can be represented in the following way:

- allocation of memory for a type;
- initialization of allocated memory (setting an object to the initial value: call of the constructor);
- the use of an object in a program;
- destruction of object's state;
- memory deallocation.

Life cycle management of an object is a tiresome task, and a constant source of errors. In order to facilitate the work with objects in C#, an automatic memory management mechanism is used called the garbage collector.

8. The concept of garbage collector

The garbage collector exempts the programmer from having to constantly monitor the use and timely release of memory.

The memory for a new object is allocated automatically by the CLR itself, using the managed dynamically allocated are of memory called the heap. Since the allocated memory is finite, it is particularly important to release the memory allocated for objects that are no longer used in an application (they do not have any valid references). This work is done by the garbage collection, which is an undetermined process; i.e. it is impossible to know in advance when the CLR «decides» to run garbage collection in your application. However, you can force garbage collection by calling the `Collect()` method of the `System.GC` class in the application.

Garbage collection consists of the following steps:

- The garbage collector searches for the managed heap objects without references.
- The garbage collector attempts to finish objects that are not referenced.
- The garbage collector releases the memory allocated for these objects.

During the program execution, the garbage collector does not release the memory allocated for an object, while it is still used by an application. In other words, is it accessible. To define the accessibility of an object, the CLR creates an object

graph, with an individual graph for each object. Object graph is a set of object graphs, each of which stores all the sets of relations of a particular object. Each graph of an object fixes the relationship of this object to another object, or its reference to another object. As long as there is at least one object graph with a reference to a specific object, the object is considered to be accesible.

It should be understood that the garbage collector will not run until the heap will have enough space for new objects, even if the majority of objects in memory are inaccessible.

9. The concept of generations in garbage collection

Since the process of availability check for all objects would take a lot of time, naturally impacting performance of an application, it was decided to divide the managed heap into three generations (0, 1, 2), and check the objects only in that generation, the amount of memory in which is critical for allocating new objects.

When all the newly created objects are allocated in the generation 0, the amount of memory of this generation is about 256 Kbytes. As soon as the amount of free memory reaches a certain level, the garbage collector will run, analyzing the objects of generation zero. All the objects without a reference will be destroyed, while the rest of the objects will be moved to generation 1. In this case, the garbage collector performs defragmentation of memory and updates the references to these objects.

In Figure 9.1, this process is presented graphically; the objects without references are grayed out (C, E, F).

In generation 1, there are some objects remained after cleaning generation 0, and the amount of memory of generation 1 is about 2 Mbytes. At some point, this amount of memory also becomes full, and the garbage collector will be called for cleaning generation 1. All the objects used by an application will move to the next generation, while the rest will be destroyed.

Generation 2 is intended to store objects left after the garbage collection process in the first generation, the amount of memory

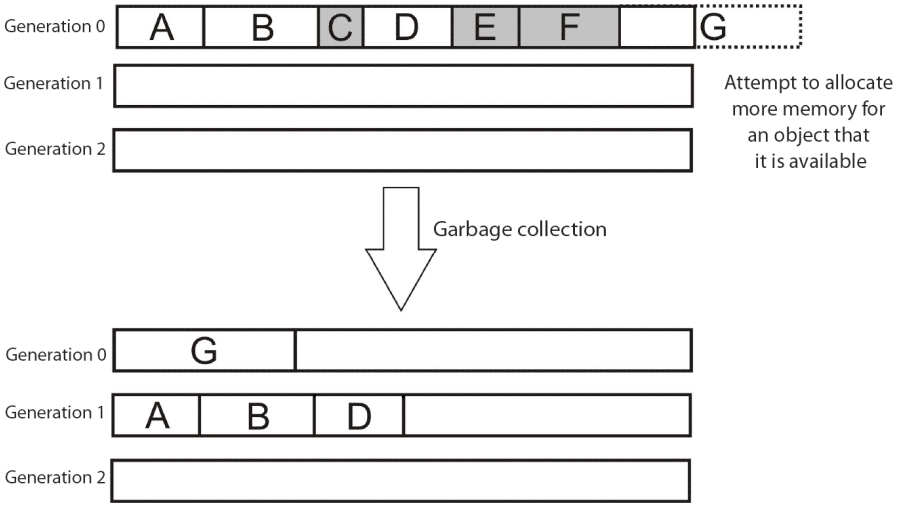

Figure 9.1. The work of the garbage collector with the generation 0 objects

of generation 2 is about 10 Mbytes. It is the latest generation, and the garbage collector will also be called if there is not enough memory for allocation of objects, but the objects have nowhere to move.

Memory sized for all generations are given, based on the information from open sources, but they are actually approximate, as they can be dynamically increased by the CLR, if necessary.

10. The System.GC class

In order to programmatically interact with the garbage collector, the `System.GC` class is provided in the .NET Framework. Basically, this class should be applied when working with unmanaged resources, while in case of managed code, the garbage collector should be called only in the most extreme cases.

The first of these cases may occur when your application begins executing some code, and its termination would lead to irreparable consequences. Garbage collection may be one of the reasons for this termination, because despite the fact that the work of garbage collector is optimized enough, when defragmenting memory, the flow termination occurs at runtime, albeit for a short time.

The second situation may occur when it is required to allocate a large amount of memory for certain objects, and should be allocated as soon as possible.

Forcing a garbage collection is performed by calling the `Collect()` method of the `System.GC` class. The overloaded version of this method accepts a generation number, in which it is necessary to perform garbage collection.

The methods of the `System.GC` class affect not only how the garbage is collected, but also when the resources allocated by an object will be released. The properties of this class provide information on the total amount of available memory and which generation belongs the memory allocated for an object.

An example of basic `System.GC` class methods is shown below (Figure 10.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Demonstration of System.GC");
            WriteLine($"Maximum generation:
                     {GC.MaxGeneration}”);
            GarbageHelper hlp = new GarbageHelper();
            //finding out a generation, in which
            //an object is located
            WriteLine($"Object generation:
                      {GC.GetGeneration(hlp)}");
            //amount of memory used
            WriteLine($"Memory used (bytes):
                      {GC.GetTotalMemory(false)}");
            hlp.MakeGarbage(); //creating some garbage
            WriteLine($"Memory used (bytes):
                     {GC.GetTotalMemory(false)}");
            GC.Collect(0); //calling explicit garbage
                           //collection in generation 0
            WriteLine($"Memory used (bytes):
                     {GC.GetTotalMemory(false)}”);
            WriteLine($"Object generation:
                     {GC.GetGeneration(hlp)}”);
            GC.Collect(); //calling explicit garbage
                          //collection in all generations
            WriteLine($"Memory used (bytes):
                      {GC.GetTotalMemory(false)}”);
            WriteLine($"Object generation:
                      {GC.GetGeneration(hlp)}”);
        }
    }
```

```
    //Additional class for garbage creation
    class GarbageHelper
    {
        //A method that creates garbage
        public void MakeGarbage()
        {
            for (int i = 0; i < 1000; i++)
            {
                Person p = new Person();
            }
        }
        class Person
        {
            string _name;
            string _surname;
            byte _age;
        }
    }
}
```

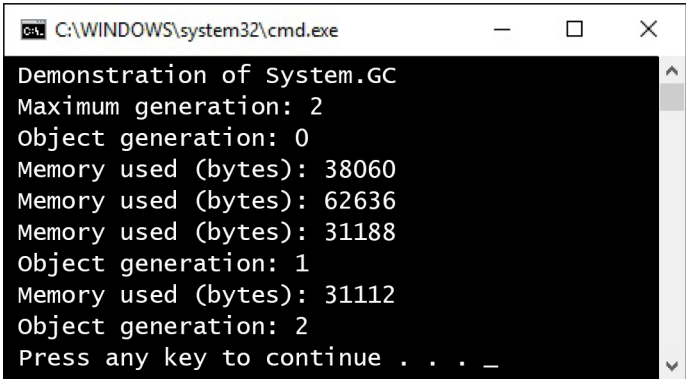

Figure 10.1. Example of working with the main methods of the System.GC class

11. Finalizer and the Finalize method

The life cycle of an object completes with the release of memory allocated when this object was created. If this object uses only managed types, then the developer has no need to create additional logic for memory deallocation, because the garbage collector will automatically release the memory allocated for them.

The CLR supports a mechanism called object finalization for the objects that use unmanaged resources. To implement this mechanism in an object, it is necessary to override the `Finalize()` method of the `System.Object` class. This method cannot be called directly through an instance of a class, as it has the `protected` access modifier. This method is called automatically by the garbage collector immediately before destroying an object.

An attempt to overload the Finalize() method using the `override` keyword will cause an error at compile time (Figure 11.1).

```
namespace SimpleProject
{
    class FinalizeExample

    {
        protected override void Finalize() { }
    }
    class Program
    {
```

```
        static void Main(string[] args)
        {
        }
    }
}
```

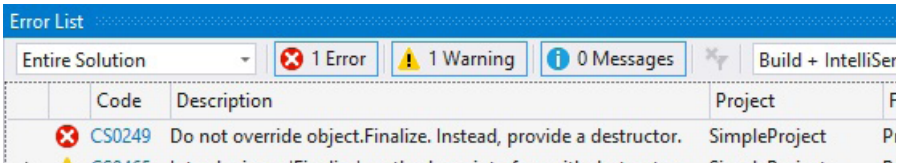

Figure 11.1. Error: do not overload the Finalize() method; create a finalizer

In fact, a special construction called finalizer is used to overload the Finalize() method. The name of a finalizer matches the name of a class, and is highlighted with a tilde (~). Finalizer cannot be overloaded; it does not have an access modifier, takes no parameters and has no return type.

```
namespace SimpleProject
{
    class FinalizeExample
    {
        ~FinalizeExample()
        { }
    }
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Unlike the C++ destructor, the finalizer does not support determined destruction of an object. And although it is

syntactically similar to the destructor of the C++, it's only a superficial resemblance. This is explained by the fact that the 1.0 version of the C# originally used a destructor. And as a legacy from C++, a destructor has been replaced by a finalized in the future versions of C#, but the syntax remained the same for version compatibility.

After compiling this code, we can see the resulting CIL code using a disassembler (Ildasm.exe), where we find that the finalizer in out `FinalizeExample` class has been replaced with the Finalize() method (Figure 11.2).

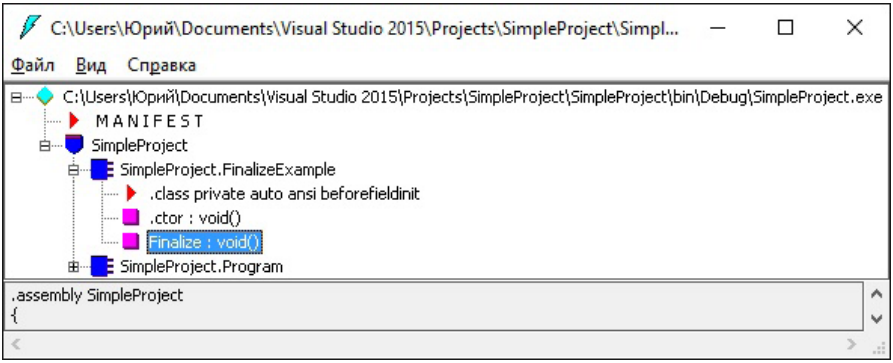

Figure 11.2. The image of the Finalize() method of the FinalizeExample type

The process of allocation and release of memory in the classes with a finalizer is also somewhat specific.

When allocating memory, each object of the heap is checked for the Finalize() method, and if it is found, then the object is marked as finalizeable, and a reference to it is stored in a special finalization queue. When the garbage collector is launched, it checks whether a current object is finalizeable and whether it has been finalized. If it has not been finalized, then the garbage collector puts this object into a special table

of objects available for finalization. Then it creates a separate flow, in which the Finalize() method is called for all the objects of this table. Meanwhile, it cleans the memory from objects that do not require finalization. And only at the next call of garbage collector, these finalizeable objects will be destroyed. Thus, it turns out that two calls of the garbage collector are required to clean the memory allocated for finalizeable objects.

Therefore, it is strongly not recommended to use a finalizer without a good reason, as the object finalization takes some additional time.

12. The Dispose method and IDisposable interface

All unmanaged objects are released by the garbage collector during finalization, and the `Finalize()` method cannot be called directly. However, the situations are common where it is necessary to release resources immediately after their use, for example, to connect to a database or file descriptors. Such behavior is implemented using the `IDisposable` interface, which has a single Dispose() method. The `IDisposable` interface can be implemented both in classes and structures, unlike the finalizer that can only be defined in classes. What do you think, why?

If an object implements the `IDisposable` interface, it notifies the user of this object that to safely finish working with it, it is necessary to call the `Dispose()` method, as it requires releasing some resources. A simple example of a class that implements the `IDisposable` interface is shown in Figure 12.1.

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        public void Dispose()
        {
            WriteLine("Releasing resources");
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            DisposeExample de = new DisposeExample();
            de.Dispose();
        }
    }
}
```

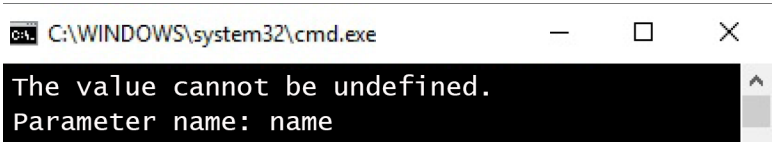

Figure 12.1. Example of a class that implements the IDisposable interface

We could continue to improve our code, but the Microsoft corporation created a universal resource cleaning template as a combination of the implementation of the `IDisposable` interface by a class and use of a finalizer in it. The use of this template allows properly cleaning machine resources both explicitly, and with automatic garbage collection (Figure 12.2).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        //it is used to find out if the Dispose()
        //method has been called
        private bool isDisposed = false
```

```
        private void Cleaning(bool disposing)
        //additional method
        {
            //make sure that the resources
            if (!isDisposed) //clean only once
            {
                //if true, then clean all
                //the managed resources
                if (disposing)
                {
                    WriteLine("Releasing managed resources");
                }

                WriteLine("Releasing unmanaged resources ");
            }
            isDisposed = true;
        }

        public void Dispose()
        {
            //calling additional method
            //true - the clean has been initialized
            //by the user of an object
            Cleaning(true);
            //forbid the garbage collector
            //to perform finalization
            GC.SuppressFinalize(this);
        }

        ~DisposeExample()
        {
            //false indicates that the cleaning
            //has been initialized by the garbage collector
            Cleaning(false);
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            DisposeExample test = new DisposeExample();
            test.Dispose();

            DisposeExample test1 = new DisposeExample();
        }
    }
}
```

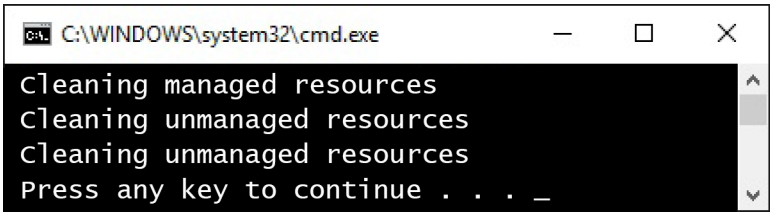

Figure 12.2. The template of the Dispose() method implementation

Note that even if there if there is an explicit management with the using the `Dispose()` method, we should also allow the implicit release of resources using the `Finalize()` method. The `Finalize()` method provides an additional way to prevent the loss of resources if the programmer forgot to call the `Dispose()` method.

13. The use of using when working with classes that implement IDisposable interface

When working with objects that implement the `IDisposable` interface, you need to ensure the cleaning of resources in case of exception, so it is necessary to use the `try-finally` block (Figure 13.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        //it is used to find out whether the Dispose()
        //method have been called
        private bool isDisposed = false;

        private void Cleaning(bool disposing)
        //additional method
        {
            //make sure that the resources have not yet
            //been released
            if (!isDisposed) //clean only once
            {
                //if true, then release all managed resources
               if (disposing)
               {
```

```
                    WriteLine("Releasing managed resources");
                }
                WriteLine("Releasing unmanaged resources");
            }
            isDisposed = true;
        }

        public void Dispose()
        {
            //calling additional method
            //true - cleaning have been initialized
            //by the user of an object
            Cleaning(true);
            //forbid garbage collector
            //to perform finalization
            GC.SuppressFinalize(this);
        }

        ~DisposeExample()
        {
            //false indicates that the cleaning has been
            //initialized by the garbage collector
            Cleaning(false);
        }

        public void DoSomething()
        {
            WriteLine("Executing certain operations");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            DisposeExample test = new DisposeExample();
```

```
            try
            {
                test.DoSomething();
            }
            finally
            {
                test.Dispose();
            }
        }
    }
}
```

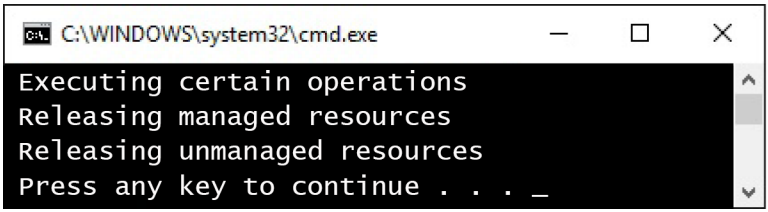

Figure 13.1. The use of the try-finally block

As an alternative of the `try-finally` block for working with classes that implement the `IDisposable` interface, we can use a special syntax with the `using` keyword.

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        //it is used to find out whether
        //the Dispose() method has been called
        private bool isDisposed = false;
```

```
private void Cleaning(bool disposing)
//additional method
{
    //make sure that the resources have
    //not been released yet
    if (!isDisposed) //clean only once
    {
        //if true, then release all managed
        //resources
        if (disposing)
        {
           WriteLine("Releasing managed resources ");
        }
        WriteLine("Releasing unmanaged resources");
    }
    isDisposed = true;
}

public void Dispose()
{
    //calling additional method
    //true - the cleaning has been initialized by
    //the user of an object
    Cleaning(true);

    //forbid the garbage collector to perform
    //finalization
    GC.SuppressFinalize(this);
}

~DisposeExample()
{
    //false indicated that the cleaning has been
    //initialized by the garbage collector
    Cleaning(false);
}
```

```
        public void DoSomething()
        {
            WriteLine("Executing certain operations");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            using (DisposeExample test =
                                    new DisposeExample())
            {
                test.DoSomething();
            }
        }
    }
}
```

Program outcome will remain the same (Figure 13.1). And if you look at the compiled CIL code using the ildasm.exe utility, then you will find out that the syntax of the `using` statement is actually converted into the `try-finally` block (Figure 13.2).

SimpleProject.Program::Main : void(string[])

Найти Найти далее

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Размер кода:       30 (0x1e)
  .maxstack  1
  .locals init ([0] class SimpleProject.DisposeExample test)
  IL_0000:  nop
  IL_0001:  newobj     instance void SimpleProject.DisposeExample::.ctor()
  IL_0006:  stloc.0
  .try
  {
    IL_0007:  nop
    IL_0008:  ldloc.0
    IL_0009:  callvirt   instance void SimpleProject.DisposeExample::DoSomething()
    IL_000e:  nop
    IL_000f:  nop
    IL_0010:  leave.s    IL_001d
  }  // end .try
  finally
  {
    IL_0012:  ldloc.0
    IL_0013:  brfalse.s  IL_001c
    IL_0015:  ldloc.0
    IL_0016:  callvirt   instance void [mscorlib]System.IDisposable::Dispose()
    IL_001b:  nop
    IL_001c:  endfinally
  }  // end handler
  IL_001d:  ret
} // end of method Program::Main
```

Figure 13.2. The IL code of the using syntax

14. Home task

Write the `Money` class for storing a sum of money (in dollars and cents). Implement overloading of the following operators: + (*adding money*), - (*subtracting money*), / (*dividing the sum by an integer*), * (*multiplying the sum by an integer*), ++ (*increase the sum by 1 cent*), - (*decrease the sum by 1 cent*), <, >, ==, !=.

The class cannot contain a negative amount of money. If execution of any operation results in a negative sum, the class throws a "Bankrupt" exception.

Using the menu, the program should demonstrate all the possibilities of the `Money` class. Exception handling is performed in the program.

Lesson 7
Exception handling. Garbage collector

© Yuriy Zaderey
© STEP IT Academy.
www.itstep.org

All rights to protected pictures, audio, and video belong to their authors or legal owners.
Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.