Exceptions

oxygen

24. oktober 2017

Agenda

oxygen

Exception handling

Conventional Exception Handling

Object-oriented Exception Handling

Exceptions and Exception Handling in C#

Exception handling

Questions about Exception Handling

oxygen

We start the lecture about exception handling.

We could as well just use the word "error handling".

Before we approach object-oriented exception handling we will in this chapter discuss error handling broadly and from several perspectives.

The motivation

oxygen

The following items summarize why we should care about error handling:

- Understand the nature of errors
  - "An error is not just an error"
- Prevent as many errors as possible in the final program
  - Automatically - via tools
  - Manually - in a distinguished testing effort
- Make programs more robust
  - A program should, while executing, be able to resist and survive unexpected situations

What is an error?

oxygen

The word "error" is often used in an undifferentiated way. We will now distinguish between errors in the development process, errors in the source program, and errors in the executing program.

Errors in the design/implementation process

- Due to a wrong decision at an early point in time - a mental flaw

Errors in the source program

- Illegal use of the programming language
- Erroneous implementation of an algorithm

Errors in the program execution - run time errors

- Exceptions - followed by potential handling of the exceptions

Errors in the development process may lead to errors in the source program.

Errors in the source program may lead to errors in the running program

What is normal? What is exceptional?

oxygen

I propose that we distinguish between "normal aspects" and "exceptional aspects" when we write a program.

Without this distinction, many real-world programs will become unwieldy.

The separation between normal aspects and exceptional aspects adds yet another dimension of structure to our programs.

In many applications and libraries, the programming of the normal aspects leads to nice and well-proportional solution.

When exceptional aspects (error handling) are brought in, the normal program aspects are polluted with error handling code.

In some situations the normal program aspects are totally dominated by exceptional program aspects.

What is normal? What is exceptional?

oxygen

Below we characterize the normal program aspects and the exceptional program aspects.

Normal program aspects

- Situations anticipated and dealt with in the conventional program flow
- Programmed with use of selective and iterative control structures

Exceptional program aspects

- Situations anticipated, but not dealt with "in normal ways" by the programmer
  - Leads to an exception
  - Recoverable via exception handling. Or non-recoverable
- Situations not anticipated by the programmer
  - Leads to an exception
  - Recoverable via exception handling. Or non-recoverable
- Problems beyond the control of the program

Example

```
public static long Factorial(int n){
    if (n == 0)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

Negative input: If *n* is negative an infinite recursion will appear. It results in a StackOverflowException .

Wrong type of input: In principle we could pass a string or a boolean to the function. In reality, the compiler will prevent us from running such a program, however.

Wrong type of multiplication: The operator may be redefined or overloaded to non-multiplication.

Numeric overflow of returned numbers: The type *long* can contain the result of 20! , but not 21! .

Memory problem: We may run out of RAM memory during the computation.

Loss of power: The power may be interrupted during the computation.

Machine failure: The computer may fail during the computation.

Sun failure: The Sun may be extinguished during the computation.

Example (cont.)

oxygen

Problem 1 should be dealt with as a normal program aspects.

Problem 2 as mentioned, is prevented by the analysis of the compiler.

Problem 3 is, in a similar way, prevented by the compiler.

Problem 4 is classified as an anticipated exceptional aspect.

Problem 4 could, alternatively, be dealt with by use of another type than long, such a BigInteger which allows us to work with arbitrary large integers. (BigInteger is not part of the .Net 3.5 libraries, however).

Problem 5 could also be foreseen as an anticipated exception.

Problem 6, 7, and 8 are beyond the control of the program. In extremely critical applications it may, however, be considered to deal with (handle) problem 6 and 7.

Example (cont.)

oxygen

Let us, briefly, address the numeric overflow problem relative to C#.

Per default, numeric overflow in integer types does not lead to exceptions in C#.

(The result of the evaluation "wraps around", a gives wrong results).

It is, however, possible to embed a piece of program in a checked{...} form, in which case numeric overflow leads to exceptions.

It is also possible to ask for another default handling of numeric overflow by use of the compiler option /checked+.

Example (cont.)

oxygen

With use of normal control structures, a different type *BigInteger*, and an iterative instead of a recursive algorithm we may rewrite the program to the following version:

```
public static BigInteger Factorial(int n){
    if (n >= 0){
        BigInteger res = 1;
        for(int i = 1; i <= n; i++)
            res = res * i;
        return res;
    }
    else
        throw new ArgumentException("n must be non-negative");
}
```

Above solves **Problem 1, 4 and 5**.

Example (cont.)

oxygen

As it appears, we wish to distinguish between normal program aspects and exceptional program aspects via the programming language mechanisms used to deal with them.

In C# and similar object-oriented languages, we have special means of expressions to deal with exceptions.

The Factorial function shown above throws an exception in case of negative input.

We distinguish between different degrees of exceptional aspects.

As a programmer, you are probably aware of something that can go wrong in your program.

Other errors come as surprises. Some error situations, both the expected and the surprising ones, should be dealt with such that the program execution survives.

Others will lead to program termination. Controlled program termination, which allows for smooth program restart, will be an important theme in this lecture.

When are errors detected?

We want to find errors as soon as possible

We identify the following error identification times:

- During design and programming - *Go for it*.
- During compilation - syntax errors or type errors - *Attractive*.
- During testing - *A lot of hard work. But necessary*.
- During execution and final use of the program
  - Handled errors - *OK. But difficult*.
  - Unhandled errors - *A lot of frustration.*

When are error detected? (cont.)

oxygen

If we are clever enough, we will design and program our software such that errors do not occur at all. However, all experience shows that this is not an easy endeavour.

Still, it is good wisdom to care about errors and exception handling early in the development process.

Static analysis of the program source files, as done by the front-end of the compiler, is important and effective for relatively early detection of errors.

The more errors that can be detected by the compiler before program execution, the better. Handling of errors caught by the compiler requires very little work from the programmers.

This is at least the case if we compare it with testing efforts…

When are errors detected? (cont.)

oxygen

Systematic test deals with sample execution of carefully prepared program fragments.

The purpose of testing is to identify errors.

Testing activities are very time consuming, but all experience indicates that it is necessary.

We devote an entire lecture to testing. We will focus on unit test of object-oriented programs.

When are errors detected (cont.)

oxygen

Finally, some errors may creep through to the end-use of the program.

Some of these errors could and should perhaps have been dealt with at an earlier point in time.

But there will remain some errors in this category. Some can be handled and therefore hidden behind the scene.

A fair amount cannot be handled. Most of the discussion today are about (handled and unhandled) errors that show up in the program at execution time.

How are errors handled?

oxygen

Assuming that we now know about the nature of errors and when they appear in the running program, it is interesting to discuss what to do about them. Here follows some possibilities.

Ignore

- False alarm - Naive

Report

- Write a message on the screen or in a log
  - Helpful for subsequent correction of the source program

Terminate

- Stop the program execution in a controlled an gentle way
  - Save data, close connections

Repair

- Recover from the error in the running program
  - Continue normal program execution when the problem is solved

How are errors handled? - Ignore

oxygen

The first option - false alarm - is of course naive and unacceptable from a professional point of view.

It is naive in the sense that shortly after we have ignored the error another error will most certainly occur.

And what should then be done?

How are errors handled? - Report

oxygen

The next option is to tell the end-user about the error.

This is naive, almost in the same way as false alarm.

But the reporting option is a very common reaction from the programmer:

"If something goes wrong, just print a message on standard output, and hopefully the problem will vanish.“

At least, the user will be aware that something inappropriate has happened.

How are errors handled? - Terminate

oxygen

The termination option is often the most viable approach, typically in combination with proper reporting.

The philosophy behind this approach is that errors should be corrected when they appear.

The sooner the better.

The program termination should be controlled and gentle, such that it is possible to continue work when the problem has been solved.

Data should be saved, and connections should be closed.

It is bad enough that a program fails "today".

It is even worse if it is impossible start the program "tomorrow" because of corrupted data.

How are errors handled? - Repair

oxygen

Repair and recovery at run-time is the ultimate approach.

We all wish to use robust and stable software.

Unfortunately, there are some problems that are very difficult to deal with by the running program.

To mention a few, just think of broken network connections, full hard disks, and power failures.

It is only in the most critical applications (medical, atomic energy, etc) that such severe problems are dealt with explicitly in the software that we construct.

Needless to say, it is very costly to built software that takes such problems into account.

Where are errors handled?

oxygen

The last fundamental question is about the place in the program where to handle errors. Should we go for local error handling, or for handling at a more remote place in the program.

- Handle errors at the place in the program where they occur
  - If possible, this is the easiest approach
  - Not always possible nor appropriate

- Handle errors at another place
  - Along the calling chain
  - Separation of concerns

Conventional Exception Handling

Exception Handling Approaches

oxygen

One way to deal with errors is to bring the error condition to the attention of the user of the program. Obviously, this is done in the hope that the user has a chance to react on the information he or she receives. Not all users can do so.

If error messages are printed to streams (files) in conventional, text based user interfaces, it is typical to direct the information to the standard error stream instead of the standard output stream.

Printing error messages

- Console.Out.WriteLine(...) or Console.Error.WriteLine(...)
- Error messages on standard output are - in general - a bad idea

Exception Handling Approaches (cont.)

oxygen

We identify the following conventional exception handling approaches:

Returning error codes

- Like in many C programs
- In conflict with a functional programming style, where we need to return data

Set global error status variables

- Almost never attractive

Raise and handle exceptions

- A special language mechanism to raise an error
- Rules for propagation of errors
- Special language mechanisms for handling of errors

Mixing normal and exceptional cases

Before we enter the area of object-oriented exception handling, and exception handling in C#, we will illustrate the danger of mixing "normal program aspects" and "exceptional program aspects".

To the right a small program, which copies a file into another file, is organized in the Main method in a C# program.

The string array passed to Main is supposed to hold the names of the source and target files.

Most readers will probably agree that the program fragment shown is relatively clear and straightforward.

```
public class CopyApp
{
    public static void Main(string[] args)
    {
        FileInfo inFile = new FileInfo(args[0]),
            outFile = new FileInfo(args[1]);
        FileStream inStr = inFile.OpenRead(),
            outStr = outFile.OpenWrite();
        int c;
        do
        {
            c = inStr.ReadByte();
            if (c != -1) outStr.WriteByte((byte)c);
        } while (c != -1);

        inStr.Close();
        outStr.Close();
    }
}
```

Mixing normal and exceptional cases

We will now care about possible issues that can go wrong in our file copy program.

Some of the error handling issues are quite realistic. Others may be slightly exaggerated with the purpose of making our points.

The important lesson to learn from the example above is that the original "normal file copying aspects" in the program almost disappears in between the error handling aspects.

```
public class CopyApp
{
    public static void Main(string[] args)
    {
        FileInfo inFile;
        do
        {
            inFile = new FileInfo(args[0]);
            if (!inFile.Exists)
                args[0] = "some other input file name";
        } while (!inFile.Exists);

        FileInfo outFile;
        do
        {
            outFile = new FileInfo(args[1]);
            if (outFile.Exists)
                args[1] = "some other output file name";
        } while (outFile.Exists);

        FileStream inStr = inFile.OpenRead(),
            outStr = outFile.OpenWrite();
        int c;
        do
        {
            c = inStr.ReadByte();
            if (c != -1) outStr.WriteByte((byte)c);
            if (StreamFull(outStr))
                DreamCommand("Fix some extra room on the disk");
        } while (c != -1);

        inStr.Close();
        if (!FileClosed(inStr))
            DreamCommand("Deal with input file which cannot be closed");

        outStr.Close();
        if (!FileClosed(outStr))
            DreamCommand("Deal with output file which cannot be closed");
    }

    /* Programming pseudo commands for the sake of this example */
    public static void DreamCommand(string str) { /* Carry out the command str */ }
    public static bool FileClosed(Stream str) { /* Return if the stream str is closed */ }
    public static bool StreamFull(Stream str) { /* Return if the stream str is full */ }
}
```

oxygen

Object-oriented Exception Handling

Object-oriented Exception Handling

oxygen

There is a solid link between object-oriented programming and exception handling.

I see two solid object-oriented contributions to error handling.

The contributions are

(1) Representation of an error as an object

(2) Classification of errors in class inheritance hierarchies.

These contributions will be explained at an overall level in this chapter.

Now we will address the same issues relative to C#.

Errors as objects

oxygen

An error is characterized by several pieces of information. It is attractive to keep these information together.

By keeping relevant error information together it becomes easier to propagate error information from one place in a program to another.

Seen in this light, it is obvious that an error should be represented as an object.

All relevant knowledge about an error

is encapsulated in an object

Errors as objects (cont.)

oxygen

Encapsulation of relevant error knowledge

- Place of occurrence (class, method, line number)
- Kind of error
- Error message formulation
- Call stack information

Transportation of the error

- From the place of origin to the place of handling
- Via a special throw mechanism in the language

An error is an object. Objects are instances of classes.

Therefore there will exist classes that describe common properties of errors.

Classification of Errors

oxygen

There are many kinds of errors:

- Fatal errors
- non-fatal errors
- system errors
- application errors
- arithmetic errors
- IO errors
- software errors
- hardware errors
- etc.

It would be very helpful if we could bring order into this mess.

Classification of Errors (cont.)

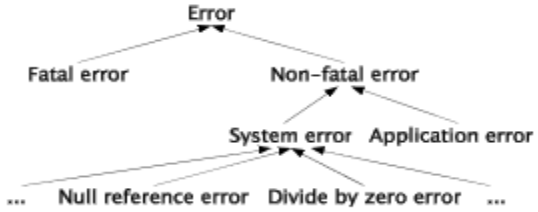
A concrete error can be represented as an instance of a class, and consequently that means we have types of errors.

Like other types, different types of errors can therefore be organized in type hierarchies.

At the programming language level we can define a set of error/classes, and we can organize these in an inheritance hierarchy.

Errors can be classified and specialized

by means of a class hierarchy



Exceptions and Exception Handling in C#

Exceptions in a C# program

oxygen

Will this program throw an exception?

```
class ExceptionDemo
{
    public static void Main()
    {
        int[] table = new int[6] { 10, 11, 12, 13, 14, 15 };
        int idx = 6;

        M(table, idx);
    }

    public static void M(int[] table, int idx)
    {
        Console.WriteLine("Accessing element {0}: {1}",
            idx, table[idx]);
    }
}
```

The try-catch statement C#

oxygen

We illustrated an unhandled exception.

The error occurred, the exception object was formed, it was propagated through the calling chain, but it was never reacted upon (handled).

We will now introduce a new control structure, which allows us to handle exceptions, as materialized by objects of type Exception.

Handling an exception imply in some situations that we attempt to recover from the error which is represented by the exception, such that the program execution can continue.

In other situations the handling of the exception only involves a few repairs or state changes just before the program terminates.

This is typically done to save data or to close connections such that the program can start again when the error in the source program has been corrected.

The try-catch statement C# (cont.)

oxygen

The try-catch statement allows us handle certain exceptions instead of stopping the program

```
try {
    try-block
}
catch (exception-type-1 name) {
    catch-block-1
}
catch (exception-type-2 name) {
    catch-block-2
}
...
```

Exercise: Handle error in table.cs

oxygen

Use a try-catch block to handle exceptions thrown in table.cs

I want you to adjust the desired index to an existing value.

The hierarchy of exceptions in C#

oxygen

The following shows an excerpt the Exception class tree in C#. The tree is shown by textual indentation. Thus, the classes ApplicationException and SystemException are sons (and subclasses) of Exception.

Exception

- ApplicationException
  - Your own exception types
- SystemException
  - ArgumentException
  - ArgumentNullException
  - ArgumentOutOfRangeException
  - DivideByZeroException
  - IndexOutOfRangeException
  - NullReferenceException
  - RankException
  - StackOverflowException
  - IOException
    - EndOfStreamException
    - FileNotFoundException
    - FileLoadException

The hierarchy of exceptions in C# (cont.)

Notice first that the Exception class tree is not the whole story. There are many more exception classes in the C# libraries than shown above.

Exceptions of type SystemException are thrown by the common language runtime (the virtual machine) if some error condition occurs.

System exceptions are nonfatal and recoverable.

As a programmer, you are also welcome to throw a SystemException object (or more precisely, an object of one of the subclasses of SystemException) from a program, which you are writing.

Originally, the exception classes that you program in your own code were intended to be subclasses of ApplicationException. In version 3.5 of the .NET framework, Microsoft recommends that your own exceptions are programmed as subclasses of Exception

Exercise Exceptions in Convert.ToDouble

oxygen

The static methods in the static class System.Convert are able to convert values of one type to values of another type.

Consult the documentation of System.Convert.ToDouble. There are several overloads of this method. Which exceptions can occur by converting a string to a double?

Write a program which triggers these exceptions.

Finally, supply handlers of the exceptions. The handlers should report the problem on standard output, rethrow the exception, and then continue.

System.Exception in C#

oxygen

The class Exception is the common superclass of all exception classes, and therefore it holds all common data and operations of exceptions.

Constructors

- Parameterless: Exception()
- With an explanation: Exception(string)
- With an explanation and an inner exception: Exception(string,Exception)

Properties

- Message: A description of the problem (string)
- StackTrace: The call chain from the point of throwing to the point of catching
- InnerException: The exception that caused the current exception
- Data: A dictionary of key/value pairs.
  - For communication in between functions along the exception propagation chain.
- *Others...*

Handling multiple types of exceptions

oxygen

There most general exception type should be at the end of the catch-clause chain

```
try{
    // Try stuff...
}
catch (NullReferenceException){
    // Handle NullReferenceException
}
catch (DivideByZeroException){
    // Handle DivideByZeroException
}
```

Handle specialized exceptions

before general exceptions

Propagation of exceptions in C#

oxygen

In the examples shown until now we have handled exceptions close to the place where they are thrown.

This is not necessary.

We can propagate an exception object to another part of the program, along the chain of the incomplete method activations.

```
public static void Main(){
    int[] table = new int[6]{10,11,12,13,14,15};
    int idx = 6;

    try{
        M(table, idx);
    }
    catch (IndexOutOfRangeException){
        int newIdx = AdjustIndex(idx,0,5);
        M(table, newIdx);
    }
    Console.WriteLine("End of Main");
}

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("Dividing by zero");
        throw;     // rethrowing the exception
    }

    Console.WriteLine("End of M");
}
```

Raising and throwing exceptions in C#

oxygen

The IndexOutOfRangeException, which we have worked with in the previous sections, was raised by the system, as part of an illegal array indexing.

We will now show how to explicitly raise an exception in our own program.

We will also see how to define our own subclass of class ApplicationException.

```
throw new CustomException("A stupid exception occured")
```

```
class CustomException : ApplicationException{
    public MyException(String problem):
        base(problem){
    }
}
```

It is recommended to adhere to a coding style where the suffixes (endings) of exception class names are "...Exception".

Try-catch with a finally clause

oxygen

A try-catch control structure can be ended with an optional finally clause.

Thus, we really deal with a try-catch-finally control structure.

```
try {
    try-block
}
catch (exception-type name) {
    catch-block
}
...
finally  {
    finally-block
}
```

At least one **catch** or **finally** clause must appear in a **try** statement.

The **finally** clause will be executed in all cases, both in case of errors, in case of error-free execution of **try** part, and in cases where the control is passed out of **try** by means of a jumping command.

Exercise – Examine finally.cs

oxygen

Look at the file finally.cs (Provide on GitHub)

Copy the code and execute in a console application. Verify that finally is always called.

Rethrowing an exception

oxygen

We will now study the idea of rethrowing an exception.

Rethrowing

- Preserving information about the original exception, and the call chain
- Usually recommended

Rethrowing is intended to forward an exception up the chain to an outer handler.

Touching, but not handling the exception

An outer handler will see the original exception

Raising an exception in a handler

oxygen

We will now study an alternative to rethrowing

Raising and throwing a new exception

- Use this approach if you, for some reason, want to hide the original exception
  - Security, simplicity, ...
- Consider propagation of the inner exception

Recommendation for exception handling

oxygen

We are now almost done with exception handling. We will now formulate a few recommendations that are related to exception handling.

Control flow

- Do not use throw and try-catch as iterative or conditional control structures
- Normal control flow should be done with normal control structures

Efficiency

- It is time consuming to throw an exception
- It is more efficient to deal with the problem as a normal program aspect - if possible

Naming

- Suffix names of exception classes with "Exception"

Exception class hierarchy

- Your own exception classes should be subclasses of Exception

Recommendation for exception handling

oxygen

Exception classes

- Prefer predefined exception classes instead of programming your own exception classes
- Consider specialization of existing and specific exception classes

Catching

- Do not catch exceptions for which there is no cure
- Leave such exceptions to earlier (outer) parts of the call-chain

Burying

- Avoid empty handler exceptions - exception burrying
- If you touch an exception without handling it, always rethrow it

oxygen

Questions?