33. Fundamental Questions about Exception Handling

With this chapter 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. In Chapter 34 we will discuss non-OO, conventional exception handling. In Chapter 35 we encounter object-oriented exception handling. Finally, in Chapter 36 we discuss exception handling in C#. Chapter 36 is the main chapter in the lecture about exception handling.

33.1. What is the motivation?

Lecture 9 - slide 2

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 be able to resist and survive unexpected situations

33.2. What is an error?

Lecture 9 - slide 3

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

33.3. What is normal? What is exceptional?

Lecture 9 - slide 4

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. This is exemplified in Section 34.2.

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

Let us assume that we program the following simple factorial function. Recall that "*n factorial*" = `Factorial(`*n*`)` = *n*! = *n* * (*n* -1) * ... * 1.

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

The following problems may appear when we run the `Factorial` function:

1. **Negative input** : If `n` is negative an infinite recursion will appear. It results in a `StackOverflowException` .
2. **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.
3. **Wrong type of multiplication** : The operator my be redefined or overloaded to non-multiplication.
4. **Numeric overflow of returned numbers** : The type `long` cannot contain the result of `20!` , but not `21!` .
5. **Memory problem** : We may run out of RAM memory during the computation.
6. **Loss of power** : The power may be interrupted during the computation.
7. **Machine failure** : The computer may fail during the computation.

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

Problem 1 should be dealt with as a normal program aspects. As mentioned, the problem in item 2 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 5, 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.

With use of normal control structures, a different (although a hypothetical) 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");
}
```

With this rewrite we have dealt with problem 1, 4, and 5. As an attractive alternative to the `if-else`, problem 1 could be dealt with by the precondition `n >= 0` of the `Factorial` method, see Section 50.1.

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. See Section 36.2 for details.

Above, 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.

33.4. When are errors detected?

Lecture 9 - slide 6

It is attractive to find errors as early as possible

Our next question cares about the point in time where you - the program developer - realize the problem. It should be obvious that we wish to identify troubles 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.*

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 endeavor. Still, it is good wisdom to care about errors and exception handling early in the development process. Problems that can be dealt with effectively at an early point in time will save a lot of time and frustrations in the latter phases of 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, described next.

Systematic test deals with sample execution of carefully prepared program fragments. The purpose of testing is to identify errors (see also Section 54.1). Testing activities are very time consuming, but all experience indicates that it is necessary. We devote a lecture, covered by Chapter 56 in this material, to testing. We will in particular focus on unit test of object-oriented programs.

Software test is also known as validation in relation to the specification of the software. Alternatively, the program may be formally verified up against a specification. This goes in the direction of a *mathematical proof*, and an area known as *model-checking*.

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 in this and the following three chapters are about (handled and unhandled) errors that show up in the program at execution time.

33.5. How are errors handled?

Lecture 9 - slide 7

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

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?

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.

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.

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 harddisks, 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.

33.6. Where are errors handled?

Lecture 9 - slide 8

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

If many errors are handled in the immediate proximity of the source of the error, chances are that a small and understandable program becomes large, unwieldy, and difficult understand. *Separation of concerns* is worth considering. One concern is the normal program aspects (see Section 33.3). Another concern is exception handling. The two concerns may be dealt with in different corners or the program. Propagation of errors from one place in a C# program to another will be discussed in Section 36.7 of this material.

34. Conventional Exception Handling

Before we approach exception handling in object-oriented programs we will briefly take a look at some conventional ways to deal with errors. You can, for instance, think of these as error handling techniques in C programming.

34.1. Exception Handling Approaches

Lecture 9 - slide 10

One way to deal with errors is to bring the error condition to the attention of the user of the program. (See Section 33.5). 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

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

When a function is used in imperative C programming, the value returned by the function can be used to signal an error condition to its caller. Many functions from the standard C library signal errors via the value returned from the function. Unfortunately, varying conventions are applied. In some functions, such as `main`, a non-zero value communicates a problem to the operating environment. In functions that return pointers (such as `malloc`) a `NULL` value typically indicates an error condition. In other functions, such as `fgetc` and `fputc`, the distinguished `EOF` (end of file) value is used as an error value. Other functions, such as `mktime`, use -1 as an error value.

As an supplementing means, a global variable can be used for signaling more details about an error. In C programming, the variable `errno` from the standard library `errno.h` is often used. When a function returns an error value (as discussed above), the value of `errno` is set to a value that gives more details about the error. Some systems use `errno` as an index to an array of error messages.

Use of error return values and global error variables is not a good solution to the error handling problem. Therefore, most contemporary languages come with more sophisticated facilities for raising, propagating and handling exceptions. The C# error handling facilities are covered specifically in Section 36.2, Section 36.3, and Section 36.7.

34.2. Mixing normal and exceptional cases

Lecture 9 - slide 11

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". See also the discussion in Section 33.3.

In Program 34.1 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 in Program 34.1 is relatively clear and straightforward.

```
using System;
using System.IO;

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();
 }
}
```

Program 34.1 *A file copy program.*

We will now care about possible issues that can go wrong in our file copy program. The result can be seen in Program 34.2, where all **red** aspects are oriented towards error handling. Some of the error handling issues are quite realistic. Others may be slightly exaggerated with the purpose of making our points.

```
1  using System;
2  using System.IO;
3  
4  public class CopyApp {
5  
6    public static void Main(string[] args) {
7      FileInfo inFile;
8      do {
9          inFile = new FileInfo(args[0]);
10         if (!inFile.Exists)
11           args[0] = "some other input file name";
12     } while (!inFile.Exists);
13 
14     FileInfo outFile;
15     do {
16         outFile = new FileInfo(args[1]);
17         if (outFile.Exists)
18           args[1] = "some other output file name";
19     } while (outFile.Exists);
20 
21     FileStream inStr   = inFile.OpenRead(),
22                outStr  = outFile.OpenWrite();
23     int c;
24     do{
25        c = inStr.ReadByte();
26        if(c != -1) outStr.WriteByte((byte)c);
27        if (StreamFull(outStr))
28          DreamCommand("Fix some extra room on the disk");
29     } while (c != -1);
30 
31     inStr.Close();
32     if (!FileClosed(inStr))
33       DreamCommand("Deal with input file which cannot be closed");
34 
35     outStr.Close();
36     if (!FileClosed(outStr))
37       DreamCommand("Deal with output file which cannot be closed");
38   }
39 }
```

Program 34.2 *A file copy program with excessive error handling.*

The line intervals 8-12 and 15-19 deal with non-existing/existing input/output files respectively. It is a problem if the input file is non-existing, and it may be problematic to overwrite an existing output file. Line 27-28 deals with memory problems, in case there is not enough room for the output file. The line intervals 32-33 and 36-37 address file closing problems. The methods `DreamCommand`, `FileClosed`, and `StreamFull` are imaginary abstractions, which are needed to complete and compile the version of the file copy program shown in Program 34.2.

The important lesson to learn from the example above is that the original "normal file copying aspects" in Program 34.1 almost disappears in between the error handling aspects of Program 34.2.

35. Object-oriented Exception Handling

It may be asked if 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, and (2) classification of errors in class inheritance hierarchies. These contributions will be explained at an overall level in this chapter. In Chapter 36 we will address the same issues relative to C#.

35.1. Errors as Objects

Lecture 9 - slide 13

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

- 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. In the next section we will discuss the organization of these classes in a class hierarchy.

35.2. Classification of Errors

Lecture 9 - slide 14

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.

In Section 35.1 we realized that a concrete error can be represented as an instance of a class, and consequently that we can deal with *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

Figure 35.1 shows an outline of type hierarchy for different kinds of errors. The concrete C# counterpart to this figure is discussed in Section 36.4.

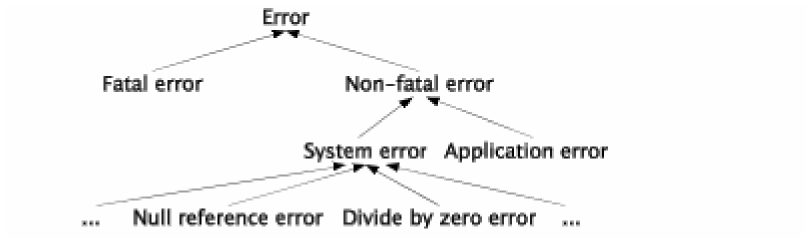

Figure 35.1 *A sample error classification hierarchy*

As hinted by the introductory words of this section, there may be several different classifications of errors. The classification in Figure 35.1 only represents one such possibility. If multiple inheritance is available (see Section 27.4 and Section 27.5) multiple error classification schemes may coexist.

36. Exceptions and Exception Handling in C#

Chapter 33, Chapter 34, and Chapter 35 have provided a context for this chapter. Warmed up in this way, we will now discuss different aspects of exceptions and exception handling in C#.

36.1. Exceptions in a C# program

Lecture 9 - slide 16

Let us start with an simple example. Several variants of the example will appear throughout this chapter. In Program 36.1 the `int` table, declared and instantiated in line 6, is accessed by the expression `M(table, idx)` in line 9. In `M` we happen to address a non-existing element in the table. Recall that an array of 6 elements holds the elements `table[0]`.. . `table[5]`. Therefore the cell addressed by `table[6]` is non-existing. Consequently, the execution of `M(table, idx)` in line 9 causes an error (index out of range). This is a *run-time error*, because the error happens when the program is executing. You can see this as a contrast to *compile-time errors*, which are identified before the program starts. In C#, a run-time error is materialized as an exception, which is an instance of class `Exception` or one of its subclasses. After its creation the exception is *thrown*. Throwing an exception means that the exception object is brought to the attention of exception handlers (catchers, see Section 36.3) which have a chance to react on the error condition represented by the exception. In the example shown in Program 36.1 the thrown exception is not handled.

```
 1 using System;
 2 
 3 class ExceptionDemo{
 4 
 5   public static void Main(){
 6     int[] table = new int[6]{10,11,12,13,14,15};
 7     int idx = 6;
 8 
 9     M(table, idx);
10   }
11 
12   public static void M(int[] table, int idx){
13     Console.WriteLine("Accessing element {0}: {1}",
14                        idx, table[idx]);
15   }
16 
17 }
```

Program 36.1 *A C# program with an exception.*

The output of Program 36.1 is shown in Listing 36.2. The effect of the `WriteLine` command in line 13 never occurs, because the error happens before `WriteLine` takes effect. The output in Listing 36.2 is therefore produced exclusively by the unhandled exception. We can see that the exception is classified as an `IndexOutOfRangeException`, which is quite reasonable. We can also see the stack trace from the beginning of the program to the place where the exception is thrown: `Main`, `M` (read from bottom to top in Listing 36.2).

```
Unhandled Exception: System.IndexOutOfRangeException:
 Index was outside the bounds of the array.
     at ExceptionDemo.M(Int32[] table, Int32 idx)
     at ExceptionDemo.Main()
```

Listing 36.2 *Output from the C# program with an exception.*

36.2. The try-catch statement C#

Lecture 9 - slide 17

In Section 36.1 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 allows us handle certain exceptions instead of stopping the program

The syntax of the new control structure is as shown below.

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

Syntax 36.1 *The syntax of try-catch statement C#*

`try-block` and `catch-block-`*i* are indeed block statements. It means that braces `{...}` are mandatory after **try** and after **catch**. Even if only a single action happens in **try** or **catch**, the action must be dressed as a block.

Let us assume that we are interested in handling the exceptions that are caused by execution of some command *c*. This can be arranged by embedding *c* in a try-catch control structure. It can also be arranged if another command *d*, which directly or indirectly activates *c*, is embedded in a try-catch control structure.

If an exception of a given type occurs, it can be handled in one of the matching catch clauses. A catch clause matches an exception object *e*, if the type of *e* is a subtype of `exception-type-i` (as given in one of the catch clauses). The matching of exceptions and catch clauses are attempted in the order provided by the catch clauses in the try control structure. Notice that each catch clause in addition specifies a name, to which the given exception object will be bound (in the scope of the handler). The names in catch clauses are similar to formal parameter names in methods.

Syntax 36.1 does not reflect the whole truth. The names of exceptions, next to the exception types, may be missing. It is even possible to have a catch clause without specification of an exception type. There is also an optional **finally** clause, which we will discuss in Section 36.9.

36.3. Handling exceptions in C#

Lecture 9 - slide 18

Now that we have introduced the try-catch control structure let us handle the exception in Program 36.1. In Program 36.3 - inside `M` - around the two activations of `WriteLine`, we introduce a try-catch construct. If an `IndexOutOfRangeException` occurs in the try part, the control will be transferred to the neighbor catch part in which we adjust the index (using the method `AdjustIndex`), and we print the result. The program now prints "`We get element number 5: 15`".

Notice that once we have left the try part, due to an error, we will not come back to the try part again, even if we have repaired the problem. In Program 36.3 this means that the `Console.WriteLine` call in 16-17 is never executed. After having executed the catch clause in line 19-23, line 24 is executed next, and `M` returns to `Main`.

```
1  using System;
2  
3  class ExceptionDemo{
4  
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8  
9      M(table, idx);
10   }
11 
12   public static void M(int[] table, int idx){
13     try{
14       Console.WriteLine("Accessing element {0}: {1}",
15                         idx, table[idx]);
16       Console.WriteLine("Accessing element {0}: {1}",
17                         idx-1, table[idx-1]);
18     }
19     catch (IndexOutOfRangeException e){
20       int newIdx = AdjustIndex(idx,0,5);
21       Console.WriteLine("We get element number {0}: {1}",
22                         newIdx, table[newIdx]);
23     }
24     Console.WriteLine("End of M");
25   }
26 
27 
28 
29   public static int AdjustIndex(int i, int low, int high){
30     int res;
31     if (i < low)
32       res = low;
33     else if (i > high)
34       res = high;
35     else res = i;
36 
37     return res;
38   }
39 }
```

Program 36.3 *A C# program with a handled exception.*

In the example above we handled the exception in the immediate neighborhood of the offending statement. It is also possible to handle the exception at a more remote place in the program, but always along the path of activated, non-completed methods. We will illustrate this in Section 36.7.

Exercise 9.1. *Exceptions in Convert.ToDouble*

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.

36.4. The hierarchy of exceptions in C#

Lecture 9 - slide 19

In this section we will take a concrete look at the classification of exceptions in C#. Our general discussion of this topic can be found in Section 35.2.

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

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.

An `ArgumentException` can be thrown if a an operation receives an illegal argument. The programmer of the operation decides which arguments are legal and illegal. The two shown subclasses of `ArgumentException` reflect that the argument cannot be `null` and that the argument is outside its legal range respectively.

The `DivideByZeroException` occurs if zero is used as a divisor in a division. The `IndexOutOfRangeException` occurs if an an array is accessed with an index, which is outside the legal bounds. The `NullReferenceExceptions` occurs in an expression like `ref.name` where ref is `null` instead of a reference to an object. The `RankException` occurs if an array with the wrong number of dimensions is passed to an operation. The `StackOverflowException` occurs if the memory space devoted to non-completed method calls is exhausted. The `IOException` (in the namespace `System.IO`) reflects different kinds of errors related to file input and file output.

`ApplicationExceptions` are "thrown when a non-fatal application error occurs" (quote from MSDN). The common runtime system throws instances of `SystemException`, not `ApplicationException`. 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` [exceptions-best-practices].

You are encouraged to identify and throw exceptions which are specializations of `SystemException`. By (re)using existing exception types, it becomes possible for the system, or for third-party program contributions, to catch the exceptions that you throw from your own code.

Exercise 9.2. *Exceptions in class Stack*

In the lecture about inheritance we specialized the abstract class `Stack`.

Now introduce exception handling in your non-abstract specialization of `Stack`. I suggest that you refine your own solution to the previous exercise. It is also possible to refine my solution.

More specifically, introduce one or more stack-related exception classes. The slide page "Raising and throwing exceptions in C#" tells you how to do it. Make sure to specialize the appropriate pre-existing exception class!

Arrange that `Push` on a full stack and that `Pop`/`Top` on an empty stack throw one of the new exceptions. Also, in the abstract stack class, make sure that `ToggleTop` throws an exception if the stack is empty, or if the stack only contains a single element.

Finally, in a sample client program such as this one, handle the exceptions that are thrown. In this exercises it is sufficient to report the errors on standard output.

Exercise 9.3. *More exceptions in class Stack*

In continuation of the previous exercise, we now wish to introduce the following somewhat unconventional handling of stack exceptions:

- If you push an element on a full stack throw half of the elements away, and carry out the pushing.
- If you pop/top an empty stack, push three dummy elements on the stack, and do the pop/top operation after this.

With these ideas, most stack programs will be able to terminate normally (run to the end).

I suggest that you introduce yet another specialization of the stack class, which specializes `Push`, `Pop`, and `Top`. The specialized stack operations should handle relevant stack-related exceptions, and delegate the real work to its superclass. Thus, in the specialized stack class, each stack operation, such as `Push`, you should embed `base.push(el)` in a **try-catch** control structure, which repairs the stack - as suggested above - in the catch clause.

36.5. The class System.Exception in C#

Lecture 9 - slide 20

The class `Exception` is the common superclass of all exception classes, and therefore it holds all common data and operations of exceptions. In this section we will examine the class `Exception` in some details.

- 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...*

`Exception` is a class (and instances of class `Exception` represents a concrete error). Therefore there exists constructors of class `Exceptions`, which (as usual) are used for initialization of a newly allocated `Exception` object. (See Section 12.4 for a general discussion of constructors).

The most useful constructor in class `Exception` takes a string, which holds an intuitive explanation of the problem. This string will appear on the screen, if a thrown exception remains unhandled. The third constructor, of the form `Exception(string,Exception)`, involves an inner exception. Inner exceptions will be discussed in Section 36.11.

As outlined above, an exception has an interface of properties (see Chapter 18) that give access to the data, which are encapsulated by the `Exception` object. You can access the message (originally provided as input to the constructor), the stack trace, a possible inner exception, and data in terms of a key-value dictionary (used to keep track of additional data that needs to travel together with the exception). For general information about dictionaries, see Chapter 46.

36.6. Handling more than one type of exception in C#

Lecture 9 - slide 21

We now continue the example from Section 36.3. In this section we will see how to handle multiple types of exceptions in a single try-catch statement.

The scene of Program 36.4 is similar to the scene in Program 36.3. In the catch clauses of the try-catch control structure we handle `NullReferenceException` and `DivideByZeroException`. On purpose, we do not yet handle `IndexOutOfRangeException`. Just wait a moment...

```
using System;

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){
    try{
      Console.WriteLine("Accessing element {0}: {1}",
                         idx, table[idx]);
    }
    catch (NullReferenceException){
      Console.WriteLine("A null reference exception");
      throw;      // rethrowing the exception
    }
    catch (DivideByZeroException){
      Console.WriteLine("Divide by zero");
      throw;      // rethrowing the exception
    }

  }
}
```

Program 36.4 *A C# program with an exception handling attempt - not a success.*

When we run the program in Program 36.4 the two handlers do not match the exception that occurs (the `IndexOutOfRangeException` exception). Therefore the exception remains unhandled, and the program stops with the output shown in Listing 36.5.

Notice that we do not provide names of the exceptions in the catch clauses in Program 36.4. We could do so. But because the names are not used they would cause the compiler to issue warnings.

While we are here, let us dwell on the two catch clauses that actually appear in the try-catch statement in Program 36.4. The null reference exception describes the problem of executing `r.f` in state where `r` refers to `null`. The divide by zero exception describes the problem of executing `a/b` in state where `b` is zero. The catch clauses report the problems, but they do not handle them. Instead, both catch clauses rethrow the exceptions. This is done by `throw` in line 19 and 23. By rethrowing the exceptions, an outer exception handler (surrounding the try catch) or exception handlers along the dynamic calling chain will have a chance to make a repair. Reporting the exception is a typical temptation of the programmer. But the reporting itself does not solve the problem! Therefore you should rethrow the exception in order to let another part of the program have to chance to make an effective repair. Rethrowing of exceptions is discussed in Section 36.10.

```
Unhandled Exception:
 System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.5 *Output from the C# program with an unhandled exception.*

It was really too bad that we did not hit the `IndexOutOfRangeException` exception in Program 36.4. In Program 36.6 we will make a better job.

We extend the catch clauses with two new entries. We add the `IndexOutOfRangeException` and we add the root exception class `Exception`. Notice that the more general exception classes should always occur at the rear end of the list of catch clauses. The reason is that the catch clauses are consulted in the order they appear. (If the `Exception` catcher was the first one, none of the other would ever have a chance to take effect).

In the concrete example, the `IndexOutOfRangeException` clause is the first that matches the thrown exception. (Notice that newly added `Exception` clause also matches, but we never get that far). Therefore we get the output shown in Listing 36.7.

```
using System;

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){
    try{
      Console.WriteLine("Accessing element {0}: {1}",
                        idx, table[idx]);
    }
    catch (NullReferenceException){
      Console.WriteLine("A null reference exception");
      throw;      // rethrowing the exception
    }
    catch (DivideByZeroException){
      Console.WriteLine("Divide by zero");
      throw;      // rethrowing the exception
    }
    catch (IndexOutOfRangeException){
      int newIdx = AdjustIndex(idx,0,5);
      Console.WriteLine("We get element number {0}: {1}",
                           newIdx, table[newIdx]);
    }
    catch (Exception){
      Console.WriteLine("We cannot deal with the problem");
      throw;      // rethrowing the exception
    }

  }

}
```

Program 36.6 *A C# program with an exception handling attempt - now successful.*

```
We get element number 5: 15
```

Listing 36.7 *Output from the C# program with a handled exception.*

Handle specialized exceptions before general exceptions

36.7. Propagation of exceptions in C#

Lecture 9 - slide 22

In the examples shown until now (see Program 36.3, Program 36.4, and Program 36.6) 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.

In Program 36.8 there is a try-catch statement in `M` and (as a new thing) also in `Main`. The local catchers in `M` do not handle the actual exception (which still is of type index out of range). The handlers in `Main` do! When the error occurs in `M`, the local exception handlers all have a chance of handling it. They do not! Therefore the exception is *propagated* to the caller, which is `Main`. Due to the propagation of the exception, line 34 of `M` is never executed. The catchers around the activation of `M` in `Main` have a relevant clause that deals with `IndexOutOfRangeException`. It handles the problem by use of the static method `AdjustIndex`. After having executed the catch clause in `Main`, the command after **`try-catch`** in `Main` is executed (line 17). The output of Program 36.8 is shown in Listing 36.9.

```
1  using System;
2  
3  class ExceptionDemo{
4  
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8  
9      try{
10       M(table, idx);
11     }
12     catch (IndexOutOfRangeException){
13       int newIdx = AdjustIndex(idx,0,5);
14       Console.WriteLine("We get element number {0}: {1}",
15                            newIdx, table[newIdx]);
16     }
17     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");
  }

}
```

Program 36.8 *A C# program with simple propagation of exception handling.*

```
We get element number 5: 15
End of Main
```

Listing 36.9 *Output from the C# program simple propagation.*

In order to illustrate a longer error propagation chain, we now in Program 36.10 introduce the calling chain

`Main` $\longrightarrow$ `M` $\longrightarrow$ `N` $\longrightarrow$ `P`

```
using System;

class ExceptionDemo{

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

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

  public static void M(int[] table, int idx){
    Console.WriteLine("M(table,{0})", idx);
    N(table,idx);
  }

  public static void N(int[] table, int idx){
    Console.WriteLine("N(table,{0})", idx);
    P(table,idx);
  }

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

}
```

Program 36.10 *A C# program with deeper exception propagation chain.*

The error occurs in `P`, and it is handled in `Main`. Here is what happens when the expression `M(table, idx)` in line 11 is executed:

1. The method `M` calls method `N` , `N` calls `P` , and in `P` an exception is thrown.
2. The error is propagated back from `P` to `Main` via `N` and `M` , because there are no (relevant) handlers in `P` , `N` or `M` .
3. The exception is handled in `Main` , and as part of the handling `M` is called again: `M(table, AdjustIndex(idx,0,5))` .
4. As above, `M` calls `N` , `N` calls `P` , and `P` calls `WriteLine` . Now no errors occur.

Due to the tracing calls of `WriteLine` in `Main`, `M`, `N`, and `P` the output shown in Listing 36.11, in part, confirms the story told about. To obtain the full confirmation, consult Exercise 9.4.

```
Main
M(table,6)
N(table,6)
P(table,6)
M(table,5)
N(table,5)
P(table,5)
Accessing element 5: 15
```

Listing 36.11 *Output from the C# program deeper exception propagation.*

Notice the *yo-yo effect* caused by the error deep in the calling chain.

Exercise 9.4. *Revealing the propagation of exceptions*

We have written a program that reveals how exceptions are propagated. In the program output, we see that the calling chain is `Main`, `M`, `N`, `P`.

The program output does not, however, reveal that the chain is followed in reverse order in an attempt to find an appropriate exception handler.

Revise the program with handlers in `M`, `N`, and `P` that touch the exception without actually handling it. The handlers should reveal, on standard output, that `P`, `N`, and `M` are passed in an attempt to locate a relevant exception handler. Rethrow the exception in each case.

36.8. Raising and throwing exceptions in C#

Lecture 9 - slide 23

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`.

The syntax of throw appears in in Syntax 36.2 and a simple example is shown next in Program 36.12. Notice the athletic metaphor behind throwing and catching.

```
throw exception-object
```

Syntax 36.2 *The syntax of exception throwing in C#*

```
...
throw new MyException("Description of problem");
...
```

Program 36.12 *A throw statement in C#.*

It is simple to define the class `MyException` as subclass of `ApplicationException`, which in turn is a subclass of `Exception`, see Section 36.4. Notice the convention that our own exception classes are subclasses of `ApplicationException`.

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

Program 36.13 *Definition of the exception class.*

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

36.9. Try-catch with a finally clause

Lecture 9 - slide 24

A try-catch control structure can be ended with an optional **finally** clause. Thus, we really deal with a try-catch-finally control structure. In this section we will study the **finally** clause.

The syntax of try-catch-finally, shown in Syntax 36.3, is a natural extension of the try-catch control structure illustrated in Syntax 36.1. `try-block`, `catch-block`, and `finally-block` are all block statements. As explained in Section 36.2 is means that braces `{...}` are mandatory after **try**, **catch**, and **finally**.

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

Syntax 36.3 *The syntax of the try-catch-finally statement C#*

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. We will now, in Program 36.14 study an example of a try-catch-finally statement.

`Main` of Program 36.14 arranges that `M` is called (in line 30) for each value in the enumeration type `Control` (line 6). Inside the static method `M` we illustrate a number of possible ways out of `M`:

1. If `reason` is `Returning` , `M` calls **return** .
2. If `reason` is `Jumping` , `M` calls **goto** which brings the control outside try-catch-finally.
3. If `reason` is `Continue` , **continue** forces the for loop to the next iteration, which is non-existing. The call of **continue** leaves the try-catch-finally abruptly.
4. If `reason` is `Breaking` , **break** breaks out of the for loop, and try-catch-finally is left abruptly.
5. If `reason` is `Throwing` , an `Exception` is thrown. The exception is "handled" locally.
6. If `reason` is 5, the expression `(Control)i` does not hit a value in the `Control` enumeration type. This causes execution and termination of the try clause, in particular the execution of `WriteLine` in line 17.

```
1  using System;
2  
3  class FinallyDemo{
4  
5    internal enum Control {Returning, Jumping, Continuing, Breaking,
6                           Throwing, Normal}
7  
8    public static void M(Control reason){
9      for(int i = 1; i <= 1; i++)  // a single iteration
10       try{
11         Console.WriteLine("\nEnter try: {0}", reason);
12         if (reason == Control.Returning) return;
13         else if (reason == Control.Jumping) goto finish;
14         else if (reason == Control.Continuing) continue;
15         else if (reason == Control.Breaking) break;
16         else if (reason == Control.Throwing) throw new Exception();
17         Console.WriteLine("Inside try");
18       }
19       catch(Exception){
20         Console.WriteLine("Inside catch");
21       }
22       finally{
23         Console.WriteLine("Inside finally");
24       }
25     finish: return;
```

```
  }

  public static void Main(){
    for(int i = 0; i <= 5; i++)
      M((Control)i);
  }
}
```

Program 36.14 *Illustration of try-catch-finally.*

The outcome of the example in Program 36.14 can be seen in the program output in Listing 36.15. So please take a careful look at it.

```
Enter try: Returning
Inside finally

Enter try: Jumping
Inside finally

Enter try: Continuing
Inside finally

Enter try: Breaking
Inside finally

Enter try: Throwing
Inside catch
Inside finally

Enter try: Normal
Inside try
Inside finally
```

Listing 36.15 *Output from the try-catch-finally program.*

As it appears, the finally clause is executed in each of the six cases enumerated above. Thus, it is not possible to bypass a finally clause of a try-catch-finally control structure. The finally clause is executed independent of the way we execute and leave the try clause.

You should place code in **finally** clauses of try-catch-finally or try-finally which should be executed in all cases, both in case or "normal execution", in case of errors, and in case of exit-attempts via jumping commands.

36.10. Rethrowing an exception

Lecture 9 - slide 25

We will now study the idea of rethrowing an exception. We have already encountered and discussed exception rethrowing in Section 36.6 (see Program 36.4).

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

In Program 36.16 we illustrate rethrowing by means of the standard example of this chapter. The situation is as follows:

1. `Main` calls `M` , `M` calls `N` , `N` calls `P` .
2. In `P` an `IndexOutOfRangeException` exception is thrown as usual.
3. On its way back the calling chain, the exception is caught in `N` . But `N` regrets, and *rethrows* the exception.
4. The exception is passed unhandled through `M` and `Main` .

```
using System;

class ExceptionDemo{

  public static void Main(){
    Console.WriteLine("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("M(table,{0})", idx);
    N(table,idx);
  }

  public static void N(int[] table, int idx){
    Console.WriteLine("N(table,{0})", idx);
    try{
      P(table,idx);
    }
    catch (IndexOutOfRangeException e){
      // Will not/cannot handle exception here.
      // Rethrow original exception.
      throw;
    }
  }

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

Program 36.16   *Rethrowing an exception.*

The output of Program 36.16 is shown in Listing 36.17. From the stack trace in Listing 36.17 it does not appear that the static method `N` actually has touched (and "smelled to") the exception. This is a main point of this example.

```
Main
M(table,6)
N(table,6)
P(table,6)

Unhandled Exception:
 System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at ExceptionDemo.P(Int32[] table, Int32 idx)
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.17 *Output from the program that rethrows an exception.*

Touching, but not handling the exception

An outer handler will see the original exception

36.11. Raising an exception in an exception handler

Lecture 9 - slide 26

We will now study an alternative to rethrowing, as discussed and illustrated in Program 36.16.

- Raising and throwing a new exception
    - Use this approach if you, of some reason, want to hide the original exception
        - Security, simplicity, ...
    - Consider propagation of the inner exception

In Program 36.18 we show a program similar to the program discussed in the previous section. Instead of rethrowing the exception in `N`, we throw a new instance of `IndexOutOfRangeException`. As can be seen in Listing 36.19 this affects the stack trace. From the outside, we can no longer see that the problem occurred in P.

```
using System;

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("M(table,{0})", idx);
    N(table,idx);
  }

  public static void N(int[] table, int idx){
    Console.WriteLine("N(table,{0})", idx);
    try{
      P(table,idx);
    }
    catch (IndexOutOfRangeException e){
      // Will not/can no handle here. Raise new exception.
      throw new IndexOutOfRangeException("Index out of range");
    }
  }

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

Program 36.18 *Raising and throwing a new exception.*

```
M(table,6)
N(table,6)
P(table,6)

Unhandled Exception: System.IndexOutOfRangeException:
 Index out of range
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.19 *Output from the program that raises a new exception.*

As a final option, we may wish to reflect, in relation to the client, that the problem actually occurred in `P`. This can be done by passing an *inner exception* to the new exception, as constructed in line 24 of Program 36.20. Notice the effect this has on the stack trace in Listing 36.21.

```
1  using System;
2  
3  class ExceptionDemo{
4  
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8      M(table, idx);
9    }
10 
11   public static void M(int[] table, int idx){
12     Console.WriteLine("M(table,{0})", idx);
13     N(table,idx);
14   }
15 
16   public static void N(int[] table, int idx){
17     Console.WriteLine("N(table,{0})", idx);
18     try{
19       P(table,idx);
20     }
21     catch (IndexOutOfRangeException e){
22       // Will not/cannot handle exception here.
23       // Raise new exception with propagation of inner exception.
24       throw new IndexOutOfRangeException("Index out of range",  e);
25     }
26   }
27 
28   public static void P(int[] table, int idx){
29     Console.WriteLine("P(table,{0})", idx);
```

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

Program 36.20 *Raising and throwing a new exception, propagating original inner exception.*

```
M(table,6)
N(table,6)
P(table,6)

Unhandled Exception: System.IndexOutOfRangeException:
 Index out of range ---> System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at ExceptionDemo.P(Int32[] table, Int32 idx)
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   --- End of inner exception stack trace ---
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.21 *Output from the program that raises a new exception, with inner exception.*

36.12. Recommendations about exception handling

Lecture 9 - slide 29

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 `ApplicationException`
    - Or alternatively (as of a more recent recommendation) of `Exception`.

- 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

36.13. References

[Exceptions-best-practices]	Best practices for handling exceptions (MSDN) http://msdn.microsoft.com/en-us/library/seyhszts.aspx