Practical Task 4.1

(Pass Task)

Submission deadline: 10:00am Monday, April 20
Discussion deadline: 10:00am Saturday, May 2

General Instructions

This practical task introduces you to ***error handling*** and covers a number of most common .NET Framework exception types that you will likely encounter as soon as you start developing larger applications. Exceptions that you will meet in practice can be thrown either by the ***runtime*** or the application's code. The latter case represents a class of exceptions that are defined by the application's developer. Some exceptions can be caused by a faulty program code, while other exceptions can result from an incorrect input that has not been accounted for in the application's code. When any of these errors occur, the system catches the error and raises an exception.

The process of generating and signalling an error is referred to as ***throwing an exception***. This is done using the '***throw***' keyword followed by a new instance of a class derived (inherited) from `System.Exception`. There are ***standard exception types*** provided by the .NET framework that you can use rather than creating ***a custom exception***.

Remember that execution failure occurs whenever a program (or its part) cannot do what it was designed to do. For example, if the OpenFile method cannot return an opened file handle to the caller, it would be considered an execution failure. Exceptions are the primary means of reporting errors in applications. Therefore, you must adhere to a number of rules when you throw or handle exceptions. Here are several basic rules that you should follow and always think about them when you work on your application.

- Report execution failures by throwing exceptions.
- Do not use exceptions for the normal ***flow of control***, if possible.
- Consider terminating the program by calling `System.Environment.FailFast` method instead of throwing an exception if your code encounters a situation where it is unsafe for further execution.
- Consider the performance implications of throwing exceptions. Throw rates above 100 per second are likely to noticeably impact the performance of most applications.
- Do not have public members that can either throw or not based on some option.
- Do not have public members that return exceptions as the return value.
- Do not throw exceptions from exception filter blocks (i.e. the '***catch***' block of the '***try-catch***' statement).
- Avoid explicitly throwing exceptions from '***finally***' blocks.

This list is by no means exhaustive and there are more rules and strategies. Remember that exception handling is an art which once you master grants you immense powers.

1. In this task, you will need to conduct a small research about every exception presented in the following list:

    a) NullReferenceException
    b) IndexOutOfRangeException
    c) StackOverflowException
    d) OutOfMemoryException
    e) InvalidCastException
    f) DivideByZeroException
    g) ArgumentException
    h) ArgumentOutOfRangeException
    i) SystemException

    In Section 9 of SIT232 Workbook, you will find a general information on how to ***catch*** and ***handle*** exceptions along with some details on the above exceptions. These exceptions are also briefly described and exemplified in the lecture notes for week 3 (you though are not allowed to copy the examples from

the lecture into your solution). However, you will still need to do search in the Internet to gather more details about their usage. We also recommend you to refer to the .NET Framework reference documentation, which is rich of various examples. Finally, do not forget to look at the references at the bottom of this task.

The expected solution for this task must consist of two parts: a program code that creates a situation which generates each of the above exceptions and a text report that explains in your own language the facts you found. As the result of your study, your report must answer the following questions and discuss the related issues for each of the above exceptions.

- What is a possible situation that leads to the exception?
- Who is in charge of throwing the exception: you as a programmer or the runtime system? In theory, should you throw exceptions of this type? Explain your answer.
- If you need to throw the exception, what details would you provide as a message to the user (caller)? What parameters would you include in the message?
- Can the exception be generally caught (and therefore handled)?
- If the exception occurs, should you generally catch this exception type or is it better to pass such exception to the user (caller)? It is not enough to say yes or no; you must provide an argument to support your answer.
- Is the exception a case when you want to avoid this exception to occur in your application in general? If so, what would be your actions as a programmer to avoid it?

Note that you must check whether all the above questions are actually relevant to a particular exception that you consider. Remember that some exceptions are reserved and are to be thrown by the runtime system, and in most cases indicate a bug. Other exceptions can be programmer-defined and report about an incorrect input data or anticipated application failures. Thus, some exceptions you can avoid by doing a proper argument checking. So, to answer these questions correctly, you will have to focus on multiple implementation aspects and potential situations associated with each exception type.

2. The following is an illustrative example of how your solution may look like. Here, we discuss the `InvalidOperationException` type of exception and give a sample code that throws it. This exception type is thrown when a method call is invalid for the object's current state. This, for example, can be caused by changing a collection (e.g. the content of an instance of the `List<T>`) while iterating it (e.g. via the '***foreach***' statement). A programmer can also throw this exception type in cases where they want to signal to their application that the requested operation through a method call cannot be performed because the object is in an invalid state. (Note that you will meet this exception in Task 4.2 soon.)

   Consider the following program code:

```
class Account
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public int Balance { get; private set; }

    public Account(string firstName, string lastName, int balance)
    {
        FirstName = firstName;
        LastName = lastName;
        Balance = balance;
    }

    public void Withdraw( int amount )
    {
        if ( amount > Balance )
        {
            throw new InvalidOperationException("Insufficient fund");
        }
        Balance = Balance - amount;
    }
}
```

The code above shows an `Account` class with properties to hold the account owner's names and account balance. The `Withdraw` method deducts the withdrawal amount from the balance, and if the withdrawal amount is higher than the balance, it throws an exception — the `InvalidOperationException`. When the exception is raised, it cancels the withdraw transaction and signals to the application that the account cannot be debited of that amount because of insufficient funds.

Thus, the `InvalidOperationException` exception is thrown in cases where the failure to invoke a method is caused by reasons other than invalid arguments. Because the `InvalidOperationException` exception can be thrown in a wide variety of circumstances, it is important to read the exception message returned by the Message property. How you handle the exception depends on the specific situation. Most commonly, however, the exception results from your own or user's incorrect actions, and therefore it can be anticipated and avoided.

3. Create a new C# Console Application project and import (replacing the existing file) the `Program.cs` file attached to this task. The main `Program` class, and specifically its `Main` method, should sequentially address every exception from the list via a block of program code (if applicable) placed inside the '***try***' section of the respective '***try-catch***' statement. For instance, our example given for the `InvalidOperationException` exception type can be written as follows:

```
try
{
      Account account = new Account("Sergey", "P.", 100);
      account.Withdraw(1000);
}
catch (InvalidOperationException exception)
{
      Console.WriteLine("The following error detected: " +
            exception.GetType().ToString()+ " with message \"" +
            exception.Message+"\"");
}
```

This code prints the following message to the terminal:

```
The following error detected: System.InvalidOperationException with message "Insufficient fund"
```

Note that the '***try***' section of the block must contain code that causes the error expected in the '***catch***' section. In our case, the `account.Withdraw(1000)` will throw the `InvalidOperationException` exception, which then will be handled in the '***catch***' section. Furthermore, if you need to add an auxiliary

class (e.g. the `Account` class in our example), then place it inside the `Program.cs` file next to the existing `Program` class. For your convenience, we provide a template of `Program.cs` containing our example.

Note that for some exceptions you do not need to provide a code example, just your answer in the report. We do not specify particular exceptions from the list because we expect you to give a clear reason for omitting this.

4. We expect you to answer the questions in the text report and write the associated code example in the `Program.cs` file. Every exception must be discussed in no more than half a page of text. Your solution should be concise and answer the relevant questions exactly. Remember that you must explain solutions in your words; that is, copying blocks of text from the Internet **is never acceptable**. Though you are allowed to paraphrase answers found in the Internet, be prepared to explain them during the one-on-one interview with your tutor.

Further Notes

- Explore Section 9 of the SIT232 Workbook to learn how exceptions interrupt the execution of code and can result in program termination; how to throw, catch and handle exceptions; what exception classes are provided by Microsoft .NET and how to define your own classes to form an exception hierarchy; and when and how to apply exceptions in your programs. The Workbook is available in CloudDeakin → Resources.
- The following links extracted from the .NET Framework reference documentation will give you more insights about error handling and some standard .NET exception types that you need to cover in the report:
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions
  - https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/exceptions
  - https://docs.microsoft.com/en-us/dotnet/api/system.invalidcastexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.dividebyzeroexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.argumentexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.argumentoutofrangeexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.systemexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.nullreferenceexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.indexoutofrangeexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.stackoverflowexception
  - https://docs.microsoft.com/en-us/dotnet/api/system.outofmemoryexception
- There are many other useful resources in the Internet that thoroughly cover this topic, for example:
  - https://www.dotnetperls.com/exception
  - https://www.c-sharpcorner.com/article/exception-handling-in-C-Sharp/
  - https://www.geeksforgeeks.org/exception-handling-in-c-sharp/

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your programs implement the required functionality. They must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit the expected code files as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their work and solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.
- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.

- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.