Exceptions Exercise

Goals:

- Practice with basic exception syntax.
- Catch exceptions generated by the CLR.
- Use a `finally` block to do cleanup.
- Examine the services offered by the `Exception` class.
- Define a custom exception type.
- Observe the effect of an uncaught exception.

Overview

Exceptions are an error notification mechanism. An exception is generated at the point an error is detected and caught and handled at a higher level.

All exception types descend from the common base class `Exception` defined in the .NET Framework Class Library. There are two interesting derived classes of `Exception`: `SystemException` and `ApplicationException`. The CLR and the library define many exceptions derived from `SystemException` which they use for common error conditions. Users can define custom exception types by deriving from `ApplicationException`. In this exercise, we will begin by using `SystemException` types to practice with the basics mechanics of `try`, `catch`, `throw`, and `finally`. After that we will define and use a custom exception type.

---

part 1- Basic try-catch

The CLR generates exceptions for many common error conditions such as an invalid array index, insufficient memory, use of a null reference, integer divide by zero, etc. In this first lab, we will practice catching an exception generated by the CLR.

steps:

1. Create a class called `ExceptionTest` and add a `Main` method. Allocate an array of 10 integers and attempt to assign a value into the array using an invalid index. Place the array access in a `try` block and place a `catch` block for an `IndexOutOfRangeException` after the try. Put a print statement in the catch block so you can verify that the handler code is being executed. Compile and run your program and observe the behavior.

---

part 2- Exception class

The root of the exception hierarchy is the class `Exception`. The `Exception` class offers several services to its clients. Here we examine two of the most useful: error message and stack trace. When the CLR creates an exception it will set the error message as appropriate. The stack trace will be filled in by the runtime when the exception is thrown.

steps:

1. Generate and catch an `IndexOutOfRangeException`. In the catch block, print out the message and stack trace from the exception object using the `Message` and `StackTrace` properties. Both the message and stack trace have type `string`. Compile and run your program and examine the output.
2. The exception class also provides a `ToString` method. Pass the exception object to `Console.WriteLine` and observe the results.

---

part 3- Multiple catch

The code in a try block may generate more than one type of exception. To facilitate handling these different conditions, a try block can have multiple catch blocks.

steps:

1. In your main program, create a try block that may generate two different types of exceptions. For the first exception type, read an array index from the user and apply it to an array. This may generate an `IndexOutOfRangeException`. For the second exception type, read two integers from the user and divide them. This may generate an `DivideByZeroException` if the denominator is zero.
2. Place two catch blocks after the try, one for `IndexOutOfRangeException` and one for `DivideByZeroException`.
3. Run the program several times and enter values that force both exceptions to occur.

---

part 4- General catch

The exception classes are all organized into an inheritance hierarchy. We have already seen one advantage of this: common services such as an error message can be provided by the base class and will then be available to all derived types. Another advantage is that a catch block specifying a base class matches that class and all derived classes. This makes it easy to catch all exceptions or specific categories of exceptions.

steps:

1. Create a try block containing code that may generate either an `IndexOutOfRangeException` or a `DivideByZeroException`.

2. After the try block, place a single catch for type `Exception`. Since `Exception` is the root of the entire hierarchy, this catch will catch all exceptions. Print out the `Message` and `StackTrace`. Run the program several times and enter values that force both exceptions to occur. Verify that the single catch works for both exception types.
3. Modify the catch to use the "general catch clause" - i.e. a catch which does not specify any type to catch. This is a convenient shorthand way to catch all exceptions; however, there is no reference to the exception object so you can not examine the message or the stack trace. Run the program several times and enter values that force both exceptions to occur. Verify that the general catch clause is executed for both exception types.

```
try
{
  ...
}
catch // general catch clause
{
   ...
}
```

---

part 5- Finally

A `try` block can have an optional `finally` block. The code in the `finally` block is always executed, regardless of how control leaves the `try` block.

One of the main uses of a finally block is to release resources. For example, a finally block is often used to close a disk file, close a database connection, close a network connection, release a lock, etc. The call to the Close or Release method is placed in a finally block to ensure it gets called even if the method terminates early by a premature return or by throwing an exception.

To make this lab realistic we need to use a resource that requires cleanup. We will use a disk file from the .NET Framework Class Library. The disk file class offers a fairly standard protocol for use: open, access, close. We will place the call to the close method in a finally block to ensure that it always gets called.

steps:

1. The `StreamWriter` class from the `System.IO` namespace provides a convenient way to write text files. The constructor takes the name of the disk file as a `string` argument. Data is written to the file using the standard set of `WriteLine` methods. The file is closed with the `Close` method. The following code contains a method that opens a disk file, writes a string to the file, and closes the file. Recode the `Write` method so it has a try block, a catch for the `IOException` that might be thrown by the `WriteLine` call, and a `finally` block containing the call to `Close`. Place a print statement in the `finally` block so you can observe it being called. Run the program.

```
class ExceptionTest
{
  static void Write(string filename, string message)
  {
          StreamWriter sw = new StreamWriter(filename);

          sw.WriteLine(message);

                  sw.Close();
          }

          static void Main()
          {
                  Write("data.txt", "Good Morning.");
          }
   }
```

---

part 6- Custom exceptions (optional)

Programmers can define custom exceptions by deriving a class from `System.ApplicationException`. Convention dictates that the name of the new class end in "`Exception`". It is also common to provide a constructor that takes a string and chains to the base class constructor to set the error message property inherited from `Exception`. In addition, programmers can add any fields or methods that are appropriate.

steps:

1. In this first step, we set up a bank account class that we will use to demonstrate custom exceptions. The bank account stores an `int` account number and a `double` for the account balance. The constructor sets the account number and the initial balance. A `Withdraw` method reduces the balance by the specified amount. Feel free to code the class yourself or take advantage of the sample code provided.

```
using System;

namespace Exceptions
{
  class SavingsAccount
  {
          private int    accountNumber;
          private double balance;

                  public SavingsAccount(int accountNumber, double balance)
                  {
                          this.accountNumber = accountNumber;
                          this.balance       = balance;
                  }

                  public void Withdraw(double amount)
                  {
                          balance -= amount;
                  }
          }

          class SavingsAccountTest
          {
                  static void Main()
                  {
                          SavingsAccount account = new SavingsAccount(12345, 2000);
```

```
                        account.Withdraw(500);
                }
        }
}
```

31. Create a custom exception type called `OverdrawnException` that can be used to indicate when the account is overdrawn. The class should derive from `System.ApplicationException`. Provide a constructor that takes a `string` error message and a `double` for the account balance. Have the constructor chain to the base class constructor to store the message. Provide a public field to store the balance.
32. Add error checking to the `Withdraw` method. If the balance goes negative after the withdrawal, throw an `OverdrawnException`. Modify the main program to place the call to `Withdraw` in a try block and include a catch for the potential exception.

---

part 7- Uncaught exception (optional)

A program might fail to catch an exception that is generated at runtime. When an exception is not caught and propagates all the way back to the beginning of the program, the language specification says the program will be terminated. However, it goes on to say "The impact of such termination is implementation-defined." In practice, you will likely see a dialog box notifying you of the uncaught exception - but the exact behavior may vary depending on the version of the CLR you are running and the type of your application (console, web, windows, etc.).

Two other points may be of interest. First, if your program is multithreaded and a thread throws an exception which is not caught, then only that thread is terminated, not the entire program. Second, the library class `AppDomain` allows programs to register to be notified when an unhandled exception occurs.

In this lab we will force an exception to occur and observe the runtime behavior when we fail to catch the exception.

steps:

1. Allocate an array of 10 integers and attempt to assign a value into the array using an invalid index. Compile and run your program and observe the behavior. If you are using Visual Studio .NET, it might be interesting to run the application both inside and outside the debugger to see if the behavior is different.