Exeption handling (*Erindid*)

Read: https://www.dotnetperls.com/catch, https://www.dotnetperls.com/try

The try-catch statement consists of a **try** block followed by one or more **catch** clauses, which specify handlers for different exceptions. When an exception is thrown, the common language runtime (CLR) looks for the **catch** statement that handles this exception.

Try - the block where we expect to have some exeption (error)

Catch – the block for dealing with the error; we can specify the error to catch. Default is all errors.

```
try
 {
     ProcessString(s);
 }

 catch (Exception e)
 {
     Console.WriteLine("{0} Exception caught.", e);
 }
```

Why?

If we have unexpected situations in our code, this can stop the execution. With exception handling we can control these situations.

Exceptions also provide detailed information about the problem.

There are many built in exceptions, list of some is here: http://www.authorcode.com/list-of-common-c-exceptions/

Exercise 1:

We've often tried int.Parse(string val) method to get a value from string. So far we've had a bad looking exception and code execution has stopped. Solve this as a method:

1. Try int.parse() on a string what is not a number, for example "*int a = int.Parse("3r");*". Investigate the exeption. What is the type of the exeption?
2. Put the parsing of the number in a try block. In catch block write "This is not a number"
3. Ask the user to enter a number. Keep on asking for the user to enter the number until they do. Catch the exception.
   Example:
   "Enter a number"
   "r5"
   "This is not a number, try again"
   "t"
   "This is not a number, try again"
   "5"

"Thanks!"

Exercise 2:

With file reading we've often had trouble with file location, lets catch this error. Do this as a method.

1) Ask the user to enter a file name (in a try block)
2) If the file name does not exist then catch FileNotFoundExeption
3) Write to the user detailed information about the error (including the file name and full path)

Tip: Use *Exeption.Message* property

Exercise 3:

Try dividing with zero.

Write a method that takes two numbers as parameter and prints out their division. If dividing with zero exeption happens then replace the zero with number 2 and print out the result.

Example:

„Divide(34,0)" -> „You cannot do that, but your number divided by 2 would be 17".

Exercise 4:

Create a list of type int and initialize it (new keyword). Do not add any items.

1) Try printing out the item with index 1. Investigate the exeption.
2) Catch this exeption.

Creating exeptions

In addition to cathing pre-defined exeptions it is possible to throw existing exeptions or create custom exeptions. Keyword is „throw".

Read: https://www.infoworld.com/article/3010009/application-development/implementing-a-custom-exception-class-in-c.html, https://www.dotnetperls.com/throw

Exercise 5:

Read: http://www.tutorialsteacher.com/csharp/custom-exception-csharp

Ask user to enter different department names. We have a rule that department names must start with letter 'd'. If they do not, then we need to throw a DepartmentNotValidExeption.

1) Create your own „DepartmentNotValidExeption" class (*extends System.Exeption class*)
2) Ask the user to enter department names
3) If the user input does not start with letter 'd' then thow 'DepartmentNotValidExeption'

Enums

Enums can be used for storing pre-defined values. Example:

```
class Program
{
```

```
enum Importance
{
    None,
    Trivial,
    Regular,
    Important,
    Critical
};

static void Main()
{
    // ... An enum local variable.
    Importance value = Importance.Critical;

    // ... Test against known Importance values.
    if (value == Importance.Trivial)
    {
        Console.WriteLine("Not true");
    }
    else if (value == Importance.Critical)
    {
        Console.WriteLine("True");
    }
}
```

Exercise 6 – This is practice for the practical test next week. Think along!

You need to create a machine for printing subway tickets. There are multiple types of tickets: gold, silver and bronze. Different ticket types have a different discount and prices.

	Gold ticket price	Silver ticket price	Bronze ticket price
Zone A: 0-30 km	5 €	3 €	3 €
Zone B: 31-60 km	7 €	4 €	4 €
Zone C: 61-80 km	9 €	5 €	5 €

All tickets must have these methods:

- Method for selling tickets which takes distance as parameter. Based on that finds the zone and price and if needed applies the discount
  Example: *silverMachine.SellTicket(50)-> „You bought a ticket from silvermachine300, for zone B and with price 4“*
    - If invalid distance is entered (not in range 0-80 km), the ticket is not sold and info should be displayed
- Method for displaying info about the sold ticket: ticket name, zone and price. This method can be called only from inside ticket selling method (not from main).

General info:

- Each ticket has a a price.
- Gold and bronze ticket have a an option for applying discount for the price. Different ticket types have a different level of discount.
    - For gold tickets the discount is 20% of the price

    - For bronze ticket the discount is 10% of the price
- Silver ticket should not have any properties or methods about discount!
- There should be a separate method for calculating the discount. Discount can only be automatically applied to every fourth gold ticket bought and to every fifth bronze ticket. This method cannot be used from main class.
- Each ticket should have a constructor which takes ticket name as parameter
- There should be an interface with all the common methods (interface can only contain public methods! Internal and private ones not)
- In main class create an object of every ticket selling machine type (one machine for gold tickets, one for silver and one for bronse). Sell 20 tickets with different prices with each machine. Tip: use a loop and a random generator for generating distances (to check false inputs too, the range for random should be 0-90 for example).

How to solve:

1. Think of what properties and methods are required, fill this table:

Class: Silver ticket	Class: Bronze ticket	Class: Gold ticket
What properties do we need? (What info does this class need to store) What methods should this class have?	What properties do we need? (What info does this class need to store) What methods should this class have?	What properties do we need? (What info does this class need to store) What methods should this class have?

2. Choose base class and derived classes
3. Implement base class, do not start with other classes before!
4. Test it, create new instances in main method (new objects)
5. If base class works, then start creating the derived classses, not before!

NB! Custom functionality can only be contained by derived classes, base class should not have it!

Imagine a car with and without ABS; adding ABS costs more. Cheaper cars do not contain ABS, it is only added to the more expensive car. If you add advanced features to base class it is like adding ABS to every car but keeping it disconnected.

Tips:

- Avoid duplication! Think how you would solve it before starting to code.
- Think of which class should be the base class and start by creating the base class. The base class should contain all methods and properties common to all the objects.
- The zone limits (between A, B, C) are the same for all ticket types, prices are different. Think how to solve it in a way that you should not create multiple checks for zones. Tip: use a list or an array, zone A price is always list[0] etc.

- The selling of the ticket is same for all the tickets; for some types there in addition applies also discount. Think how to solve this without code duplication. Tip: use an additional method for keeping the common logic.
- Properties/fields for derived classes should be overwritten in the constructor; not defined again.
- Class inheritance can also go A->B->C->D
- Think of which methods shlould be public and which ones not.