

# Exercise: Exception Handling

This document defines the **exercise assignments** for the ["C# OOP" course @ Software University](#).

## Problem 1. Square Root

Write a program that reads an integer **number** and **calculates** and **prints** its **square root**. If the number is invalid or negative, print "Invalid number". In all cases finally print "Good bye". Use **try-catch-finally**.

## Problem 2. Enter Numbers

Write a method **ReadNumber(int start, int end)** that enters an integer number in a given range [**start...end**]. If an **invalid number** or a **non-number** text is entered, the method should **throw an exception**. Using this method write a program that enters **10 numbers:  $a_1, a_2, \dots, a_{10}$ , such that  $1 < a_1 < \dots < a_{10} < 100$** . If the user enters an invalid number, make the user enter all of them again.

## Problem 3. Fixing

This program is throwing an **IndexOutOfRangeException**. Using your skills, fix this problem using a try catch block.

```
static void Main(string[] args)
{
    string[] weekdays = new string[5];
    weekdays[0] = "Sunday";
    weekdays[1] = "Monday";
    weekdays[2] = "Tuesday";
    weekdays[3] = "Wednesday";
    weekdays[4] = "Thursday";

    for (int i = 0; i <= 5; i++)
    {
        Console.WriteLine(weekdays[i].ToString());
    }
    Console.ReadLine();
}
```

## Problem 4. Fixing Vol2

The given program is throwing an **OverflowException**. Fix it.

```
static void Main(string[] args)
{
    int num1, num2;
    byte result;

    num1 = 30;
    num2 = 60;
    result = Convert.ToByte(num1 * num2);
    Console.WriteLine("{0} x {1} = {2}", num1, num2, result);
    Console.ReadLine();
}
```

## Problem 5. Convert.ToDouble

The static methods in the static class **System.Convert** can convert values from one type to another type. Check the documentation of **System.Convert.ToDouble**. There are several **overloads** of this **method**. Which **exceptions** can occur by converting from **string** to **double**? Write a program which triggers these exceptions.

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

## Problem 6. Valid Person

Define a simple class **Person**, which has the following fields: **first name**, **last name** and **age**. **Validate** the data in the properties' **setters**, **throw** appropriate **exceptions** in case invalid data is entered.

### Step 1. Create a Class Person

Create a project for this exercise and add a class **Person** in a separate .cs file. The class should contain the following fields: **first name (string)**, **last name (string)** and **age (int)**.

All fields are **required**, meaning you should have one constructor accepting all three as **parameters**. For example:

```
namespace ExceptionHandling_Exercises
{
    public class Person
    {
        private string firstName;
        private string lastName;
        private int age;

        public Person(string firstName, string lastName, int age)
        {
            // TODO: add properties and validate data
        }

        //TODO: add properties
    }
}
```

### Step 2. Add Properties and Validate the Data

Add a **property** for each of the fields. Perform validations in their **setters** to keep the state of the **Person** objects correct.

The **first** and **last name** cannot be **null** or **empty** strings. To check this, use the **string.IsNullOrEmpty()** method.

The **age** must be in the range **[0 ... 120]**.

If invalid data is entered, **throw** appropriate exceptions with descriptive **messages**. E.g., if an empty name is entered, an appropriate exception may be **ArgumentNullException**. If the age is negative or too big, an appropriate exception would be **ArgumentOutOfRangeException**.

Example for validating the **first name** (last name is analagous):

```
public string FirstName
{
    get
    {
        return this.firstName;
    }

    set
    {
        if (string.IsNullOrEmpty(value))
        {
            throw new ArgumentNullException(
                "value",
                "The first name cannot be null or empty.");
        }

        this.firstName = value;
    }
}
```

Example for validating the **age**:

```
public int Age
{
    get
    {
        return this.age;
    }

    set
    {
        if (value < 0 || 120 < value)
        {
            throw new ArgumentOutOfRangeException(
                "value",
                "Age should be in the range [0 ... 120].");
        }

        this.age = value;
    }
}
```

Now the constructor should make use of the properties instead of modifying the private fields directly:

```
public Person(string firstName, string lastName, int age)
{
    this.FirstName = firstName;
    this.LastName = lastName;
    this.Age = age;
}
```

## Step 3. Test the Person Class

In your main program, test whether your class behaves properly. Create several objects of type Person – one with **valid data**, one with an **empty first name**, one with **null as last name**, one with **negative age** and one **with age > 120**. Check whether executing the code results in errors, when bad data is provided. Test the invalid cases one by one by commenting out the other invalid lines of code (your program will stop executing when the first error is encountered).

```

public static void Main()
{
    Person pesho = new Person("Pesho", "Peshev", 24);

    Person noName = new Person(string.Empty, "Goshev", 31);
    Person noLastName = new Person("Ivan", null, 63);
    Person negativeAge = new Person("Stoyan", "Kolev", -1);
    Person tooOldForThisProgram = new Person("Iskren", "Ivanov", 121);
}

```

## Step 4. Add Try-Catch Blocks

To prevent the program from blowing up, surround the invalid lines in **try-catch** blocks. It's a good practice to put different catch blocks for the different types of errors you anticipate the operation might throw. Print the **message** of the exception in the catch block.

Example (invalid **name**):

```

try
{
    Person noName = new Person(string.Empty, "Goshev", 31);
}
catch (ArgumentNullException ex)
{
    Console.WriteLine("Exception thrown: {0}", ex.Message);
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine("Exception thrown: {0}", ex.Message);
}

// Result in console:
// Exception thrown: The first name cannot be null or empty.
// Parameter name: value

```

Example (invalid **age**):

```

try
{
    Person negativeAge = new Person("Stoyan", "Kolev", -1);
}
catch (ArgumentNullException ex)
{
    Console.WriteLine("Exception thrown: {0}", ex.Message);
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine("Exception thrown: {0}", ex.Message);
}

// Result in console:
// Exception thrown: Age should be in the range [0 ... 120].
// Parameter name: value

```

## Problem 7. Custom Exception

Create **InvalidPersonNameException** class in the previous problem, which does not allow any special character or numeric value in a name of any of the students. To do that create Student class with Name and Email properties. When trying to create student with name "Gin4o", throw your custom Exception class.