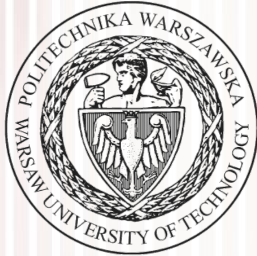
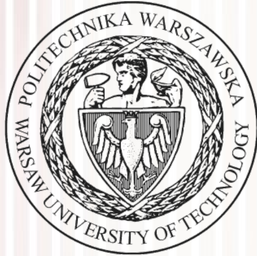


Advanced Programming C#

Lecture 8

dr inż. Małgorzata Janik
malgorzata.janik@pw.edu.pl

Winter Semester 2023/2024



TODAY

Exceptions
Func, Action delegates
Lambda expressions

Exceptions

Exception

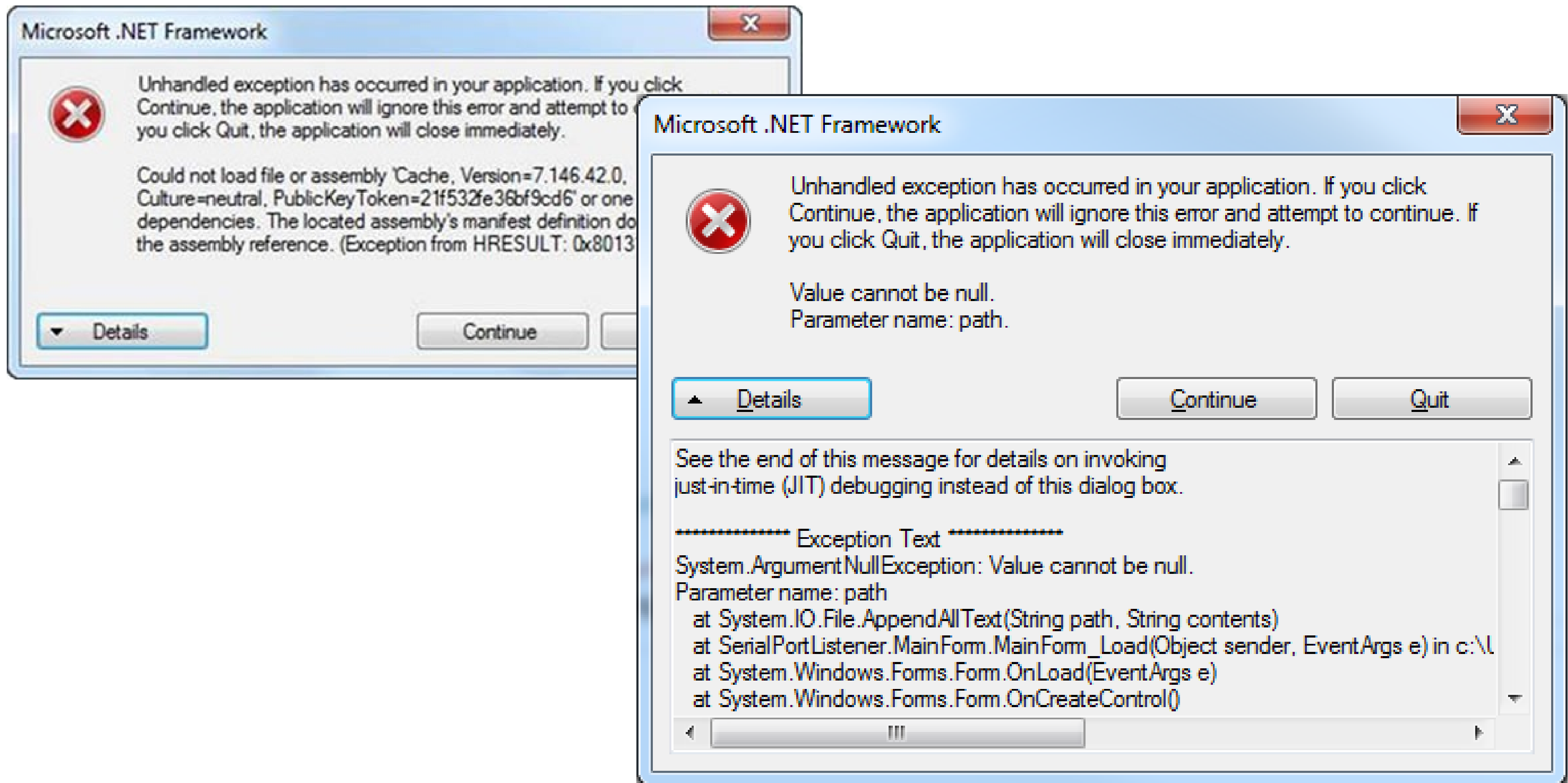
- In C#, errors in the program at run time are propagated through the program by using a mechanism called **exceptions**.
- Exceptions are thrown by code that encounters an error and caught by code that can correct the error.
- Exceptions can be thrown
  - by the .NET Framework common language runtime
  - by code in a program.

Once an exception is thrown, it propagates up the call stack until a catch statement for the exception is found.

https://msdn.microsoft.com

Exception

- Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.



https://msdn.microsoft.com

Handling exceptions

Use exception handling code (try/catch blocks) appropriately. You can also programmatically check for a condition that is likely to occur without using exception handling.

- **Programmatic checks**. The following example uses an if statement to check whether a connection is closed. If it isn't, the example closes the connection instead of throwing an exception.

```
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}
```

- **Exception handling**. The following example uses a try/catch block to check the connection and to throw an exception if the connection is not closed.

```
try
{
    conn.Close();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

https://msdn.microsoft.com

Handling exceptions

- The method you choose depends on how often you expect the event to occur.
  - Use exception handling if the event doesn't occur very often, that is, **if the event is truly exceptional and indicates an error**
  - Use the programmatic method to check for errors if the event happens **routinely** and could be considered **part of normal execution**.

https://msdn.microsoft.com

Handling exceptions

Programmers should throw exceptions when one or more of the following conditions are true:

- The method cannot complete its defined functionality.
- An inappropriate call to an object is made, based on the object state.
- When an argument to a method causes an exception.

https://msdn.microsoft.com

Handling exceptions

Programmers should throw exceptions when one or more of the following conditions are true:

- The method cannot complete its defined functionality.
  - For example, if a parameter to a method has an invalid value:

```
static void CopyObject(SampleClass original)
{
    if (original == null)
    {
        throw new System.ArgumentException("Parameter cannot be null", "original");
    }
}
```

- An inappropriate call to an object is made, based on the object state.
- When an argument to a method causes an exception.

https://msdn.microsoft.com

Handling exceptions

Programmers should throw exceptions when one or more of the following conditions are true:

- The method cannot complete its defined functionality.
- An inappropriate call to an object is made, based on the object state.
  - One example might be trying to write to a read-only file. In cases where an object state does not allow an operation, throw an instance of InvalidOperationException or an object based on a derivation of this class.

```
class ProgramLog
{
    System.IO.FileStream logFile = null;
    void OpenLog(System.IO.FileInfo fileName, System.IO.FileMode mode) {}

    void WriteLog()
    {
        if (!this.logFile.CanWrite)
        {
            throw new System.InvalidOperationException("Logfile cannot be
                                                         read-only");
        }
        // Else write data to the log and return.
    }
}
```

- When an argument to a method causes an exception.

Handling exceptions

Programmers should throw exceptions when one or more of the following conditions are true:

- The method cannot complete its defined functionality.
- An inappropriate call to an object is made, based on the object state.
- When an argument to a method causes an exception.
  - In this case, the original exception should be caught and an ArgumentException instance should be created. The original exception should be passed to the constructor of the ArgumentException as the InnerException parameter:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
    System.ArgumentException argEx = new
System.ArgumentException("Index is out of range", "index", ex);
        throw argEx;
    }
}
```

Handling exceptions

- Exceptions contain a property named StackTrace. This string contains the name of the methods on the current call stack, together with the file name and line number where the exception was thrown for each method. A StackTrace object is created automatically by the common language runtime (CLR) from the point of the throw statement, so that exceptions must be thrown from the point where the stack trace should begin.
- All exceptions contain a property named Message. This string should be set to explain the reason for the exception

Things to Avoid When Throwing Exceptions

The following list identifies practices to avoid when throwing exceptions:

- Exceptions should not be used to change the flow of a program as part of ordinary execution. Exceptions should only be used to report and handle error conditions.
- Exceptions should not be returned as a return value or parameter instead of being thrown.
- Do not throw System.Exception, System.SystemException, System.NullReferenceException, or System.IndexOutOfRangeException intentionally from your own source code.

https://msdn.microsoft.com/en-us/library/ms173163.aspx

Exceptions in reading files

```
StreamWriter sw = null;
try
{
    sw = File.AppendText(filePath);
    sw.WriteLine(message);
}
catch (IOException ex){
    MessageBox.Show(ex.Message);
}
catch(Exception ex){
    MessageBox.Show(ex.Message);
}
finally {
    if (sw != null)
        sw.Dispose();
}
```

or

```
try
{
    using (StreamWriter sw =
File.AppendText(filePath))
    {
        sw.WriteLine(message);
    }
}
catch(Exception ex)
{
   // Handle exception
}
```

Delegates: Func & Action

Different Flavors of Delegate

- Func
- Action
- Predicate
- Convert
- Comparsion

http://cezarywalenciuk.pl/blog/programing/post/c-delegaty-action-i-func

https://www.codeproject.com/articles/741064/delegates-its-modern-flavors-func-action-predicate

Different Flavors of Delegate

- Func
- Action
- Predicate
- Convert
- Comparsion

http://cezarywalenciuk.pl/blog/programing/post/c-delegaty-action-i-func

https://www.codeproject.com/articles/741064/delegates-its-modern-flavors-func-action-predicate

Delegate (once again)

- Delegate is a pointer to a method.
- Delegate can be passed as a parameter to a method.
- We can change the method implementation dynamically at run-time – only thing we need to follow doing so would be to maintain the parameter type & return type.

```
public class Game
{
    public string Name { get; set; }
    public int Year { get; set; }
}

public class Methods
{
  public string CalculateAgeEra(int year)
  {
    int currentyear = DateTime.Now.Year - year;

    if (currentyear < 2)
      return "PS4 XBOXONE Era";
    else if (currentyear < 10)
      return "PS3 XBOX Era";
    else if (currentyear < 17)
      return "PS2 GameCube Era";
    else if (currentyear < 21)
      return "PS1 Nintentdo 64 Era";
    else if (currentyear < 31)
      return "Nes Amiga Era";
    else
      return "Pegasus Era";
  }
```

string name(int)

```
  public string UpperCaseName(string s)
  {
     return s.ToUpperInvariant();
  }
```

string name(string)

```
  public string LowerCaseName(string s)
  {
     return s.ToLowerInvariant();
  }
```

string name(string)

```
  public void Show(string p1, string p2)
 {
   Console.ForegroundColor = ConsoleColor.White;
   Console.Write(p1);
   Console.ForegroundColor = ConsoleColor.Yellow;
   Console.Write(" : ");
   Console.ForegroundColor = ConsoleColor.Green;
   Console.Write(p2);
   Console.ForegroundColor = ConsoleColor.Gray;
}
```

void name(string,string)

```
 public void Show2(string p1, string p2)
 {
   Console.ForegroundColor = ConsoleColor.Yellow;
   Console.Write(p1);
   Console.ForegroundColor = ConsoleColor.DarkGreen;
   Console.Write(" : ");
   Console.ForegroundColor = ConsoleColor.Magenta;
   Console.Write(p2);
   Console.ForegroundColor = ConsoleColor.Gray;
} }
```

void name(string,string)

```
class Program {

public delegate string calculateAgeEraFunctionPointer(int year); //defincja
public delegate string changeTitleFunctionPointer(string name); //defincja del
public delegate void showTwoStringsPointer(string p1, string p2); //defincja d


static void Main(string[] args) {
Methods methods = new Methods();
calculateAgeEraFunctionPointer funcPointer = methods.CalculateAgeEra;

string f = "";
while (!(f == "1" || f == "2"))
{
      Console.WriteLine("Wybierz ukazanie tekstu");
      Console.WriteLine("1. Biało zielony");
      Console.WriteLine("2. Zółto różowy");
      f = Console.ReadKey().KeyChar.ToString();
}
Console.Clear();
showTwoStringsPointer show;
if (f == "1") show = methods.Show;
else show = methods.Show2;

f = "";
while (!(f == "1" || f == "2"))
{
      Console.WriteLine("Wybierz ukazanie wielkość tekstu");
      Console.WriteLine("1. Duże litery");
      Console.WriteLine("2. Małe litery");
      f = Console.ReadKey().KeyChar.ToString();
}
Console.Clear();
changeTitleFunctionPointer titlePointer;
if (f == "1") titlePointer = methods.UpperCaseName
else titlePointer = methods.LowerCaseName;

RestOfTheProgram(funcPointer,titlePointer,show);
}
```

string name(int)

string name(string)

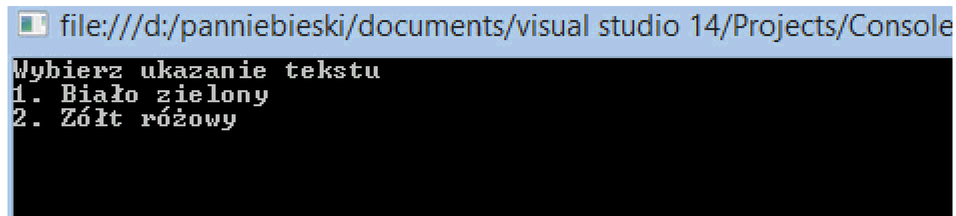
void name(string,string)

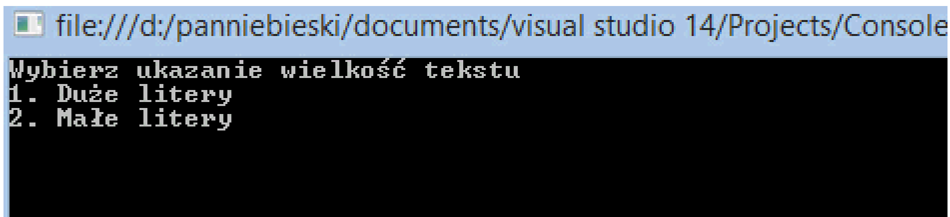
CalculateAgeEra
string name(int)

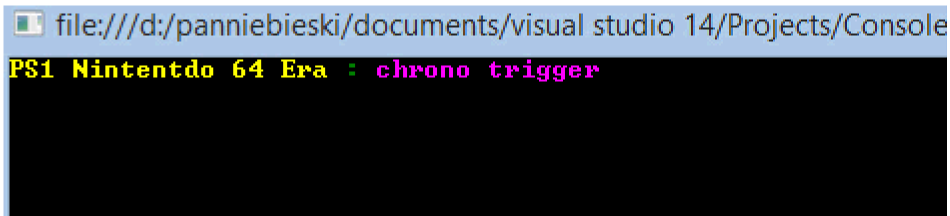
Show:
void name(string,string)

Upper/LowerCase
string name(string)

```
public static void RestOfTheProgram(calculateAgeEraFunctionPointer funcPointer,
changeTitleFunctionPointer titlePointer,
showTwoStringsPointer show)
{
    Game game = new Game() { Name = "Chrono Trigger", Year = 1995 };
    string era = funcPointer(game.Year);
    string name = titlePointer(game.Name);
    show(era, name);
    Console.ReadKey();
}
```







Action<TParameter>

- Action is similar to base delegate implementation when we do not have any return type from method - method with **void** signature.
- The difference is in the way we declare.
- At the time of declaration, we need to provide the type of the parameters:
  - `Action<string, int> tempActionPointer;`
    - the first two parameters are the method input parameters; since we do not have return object or type, all the parameters are considered as input parameters.

```
Action<string, int> tempActionPointer = tempObj.FirstTestFunction;
tempActionPointer("Hello", 8);
Console.ReadKey();
```

Func<TParameter, TOutput>

- At the time of declaration, we need to provide the signature parameter & its return type.
- **`Func<string, int, int> tempFuncPointer;`**
- First two parameters are the method input parameters. 3rd parameter (always the last parameter) is the out parameter which should be the output return type of the method.

```
Func<string, int, int> tempFuncPointer = tempObj.FirstTestFunction;
int value = tempFuncPointer("hello", 3);
Console.ReadKey();
```

- Func is always used when you have return object or type from method. If you have void method, you should be using Action

```
static void Main(string[] args)
{
Methods methods = new Methods();
Func<int, string> funcPointer = methods.CalculateAgeEra;
Func<string, string> titlePointer;
Action<string, string> show;
[...]
if (f == "1")
     show = methods.Show;
else
     show = methods.Show2;
[...]
if (f == "1")
    titlePointer = methods.UpperCaseName;
else
    titlePointer = methods.LowerCaseName;

RestOfTheProgram(funcPointer,titlePointer,show);
}
```

takes **int** return **string**

takes **string** return **string**

takes **string,string** return **void**

```
public static void RestOfTheProgram(Func<int,string> funcPointer,
Func<string, string> titlePointer,
Action<string,string> show)
{
     Game game = new Game() { Name = "Chrono Trigger", Year = 1995 };
     string era = funcPointer(game.Year);
     string name = titlePointer(game.Name);
     show(era, name);
     Console.ReadKey();
}
```

Much easier to read and understand, since it is visible what parameters / return types are connected to Func and Action

More examples

```
public class Methods {
public bool CheckifGameExist(string name,int year)
{
     return true;
}
```

More examples

```
public class Methods {
public bool CheckifGameExist(string name,int year)
{
     return true;
}
```

<u>In Main:</u>

```
Func<string, int, bool> gameExistPointer = methods.CheckifGameExist;
bool result = gameExistPointer("Kombat ", 1992);
```

More examples

```
public class Methods {
public bool CheckifGameExist(string name,int year)
{
     return true;
}
```

In Main:

```
Func<string, int, bool> gameExistPointer = methods.CheckifGameExist;
bool result = gameExistPointer("Kombat ", 1992);
```

```
public class Methods {
public void AddEnter()
{
     Console.WriteLine();
}
```

More examples

```
public class Methods {
public bool CheckifGameExist(string name,int year)
{
     return true;
}
```

In Main:

```
Func<string, int, bool> gameExistPointer = methods.CheckifGameExist;
bool result = gameExistPointer("Kombat ", 1992);
```

```
public class Methods {
public void AddEnter()
{
     Console.WriteLine();
}
```

In Main:

```
Methods methods = new Methods();
Action action = methods.AddEnter;
```

More examples

```
public class Methods {
public bool CheckifGameExist(string name,int year)
{
      return true;
}
```

In Main:

```
Func<string, int, bool> gameExistPointer = methods.CheckifGameExist;
bool result = gameExistPointer("Kombat ", 1992);
```

```
public class Methods {
public void AddEnter()
{
      Console.WriteLine();
}
```

In Main:

```
Methods methods = new Methods();
Action action = methods.AddEnter;
```

```
public class Methods {
public void AddEnterAndSomethin(string som)
{
      Console.WriteLine(); Console.WriteLine(som);
}
```

More examples

```
public class Methods {
public bool CheckifGameExist(string name,int year)
{
     return true;
}
```

In Main:

```
Func<string, int, bool> gameExistPointer = methods.CheckifGameExist;
bool result = gameExistPointer("Kombat ", 1992);
```

```
public class Methods {
public void AddEnter()
{
     Console.WriteLine();
}
```

In Main:

```
Methods methods = new Methods();
Action action = methods.AddEnter;
```

```
public class Methods {
public void AddEnterAndSomethin(string som)
{
     Console.WriteLine(); Console.WriteLine(som);
}
```

In Main:

```
Methods methods = new Methods();
Action<string> action2 = methods.AddEnterAndSomethin;
```

Func & Action

- How many parameters Func/Action can take?
  - Up to 17

```
Func<string, string> titlePointer;
```

```
Func<
▲ 17 of 17 ▼ Func<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8, in T9, in T10, in T11, in T12, in T13, in T14, in T15, in T16, out TResult>
Encapsulates a method that has 16 parameters and returns a value of the type specified by the TResult parameter.
```

    - If you need a function that takes more than 17 parameters, than there is probably something wrong with your code

http://cezarywalenciuk.pl/blog/programing/post/c-delegaty-action-i-func

Lambda expressions

Lambda expressions

- A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates (or expression tree types).
- All lambda expressions use the lambda operator **=>**, which is read as "goes to".
  - The left side of the lambda operator specifies the input parameters (if any)
  - the right side holds the expression or statement block.
- The lambda expression x => x * x is read "x goes to x times x." This expression can be assigned to a delegate type as follows:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

(input parameters) => expression

https://msdn.microsoft.com/pl-pl/library/windows/desktop/bb397687(v=vs.90).aspx

Lambda expressions

- Expression Lambdas
  - (input parameters) => expression
  - The parentheses are optional only if the lambda has one input parameter; otherwise they are required.
    - (x, y) => x == y
  - Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:
    - (int x, string s) => s.Length > x
  - Specify zero input parameters with empty parentheses:
    - () => SomeMethod()

https://msdn.microsoft.com/pl-pl/library/windows/desktop/bb397687(v=vs.90).aspx

Lambda expressions

- Statement Lambdas
  - (input parameters) => {statement;}
  - The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
delegate void TestDelegate(string s);
…
TestDelegate myDel = n => { string s = n + " " +
"World"; Console.WriteLine(s); };
myDel("Hello");
```

https://msdn.microsoft.com/pl-pl/library/windows/desktop/bb397687(v=vs.90).aspx

Lambda expressions

- Lambdas with the Standard Query Operators
  - Many Standard query operators have an input parameter whose type is one of the Func<T,TResult> family of generic delegates.

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4); // returns false of course
```

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
```

```
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
```

https://msdn.microsoft.com/pl-pl/library/windows/desktop/bb397687(v=vs.90).aspx

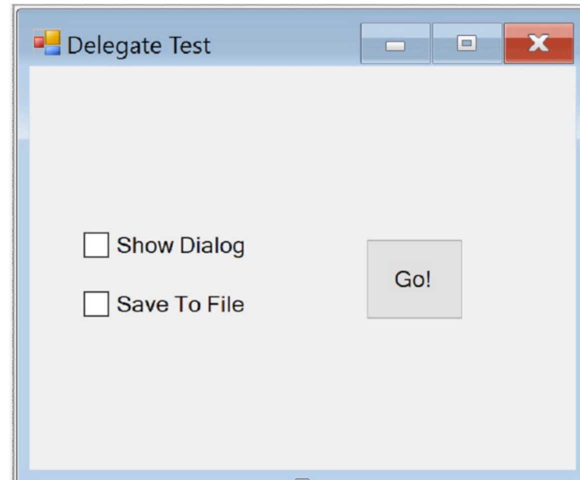
Task

Task

- **0.** Start new Windows Forms project. Default class with window is our GUI.
- **1.** Exceptions
  - Create **Output** class that includes two static methods which take string as argument:
  - **ShowDialogBox** – shows dialog box with text given as function parameter
  - **SaveToFile** – appending to results.txt file text given as function parameter
  - Remember to catch exception when saving to the file!

Task

- **2.** Delegates
  - Add two checkboxes to the main window, as below:

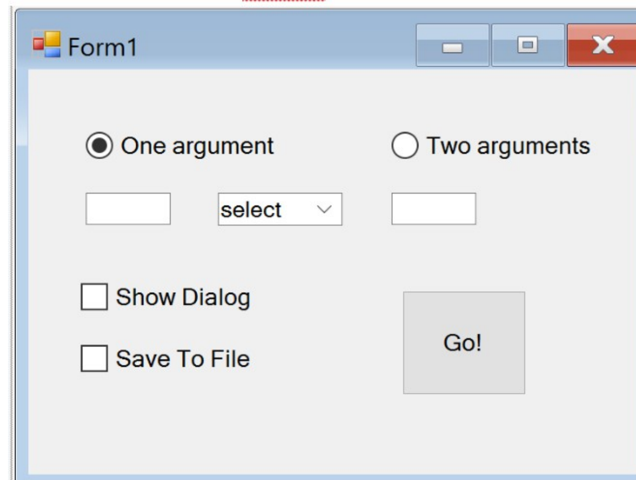


  - Write code with following functionality: depending on the checkbox value, text "**Result:** " should be (a) displayed in dialog box, (b) saved to file, (c) both options at the same time, (d) when none of options is checked, 'Go!' button is disabled.
  - Use delegate
    - `public delegate void ReturnResult(string text);`

  Which will delegate task of showing results to appropriate functions, and which will be called when the button is clicked. ⟶ You should show the code with the delegate to the lecturer, since in the next step you will change it to Action.

Task

- Action delegate
  - You should replace previously created delegate for Action with the same name.
  - Use of Action makes code easier to read (you can see function parameters immidately).
- Func and lambda expressions
  - You should add two radiobuttons, two textboxes and one combobox to the form, as shown below:



Task

- RadioButtons should work in the following way:
  - when "One argument" is checked, only one textbox should be visible, and combobox should contain following list of items: sqrt, factorial
  - when "Two arguments" is checked, both textboxes should be visible, and combobox should contain following items: +, -, *, /
- Create **two Func delegates** (eg. operationOne, operationTwo) which take two or three arguments, respectively (of type double).
- These delegates should be set each time a function is chosen from the combobox – always using **lambda expressions**. Write **factorial** yourself (statement lambda).
- **Go! button** click method should call delegates operationOne or operationTwo (depending which option is active)

Additional info

Delegate vs Interface

- It is always possible to use interface instead of a delegate.
- When to use delegates?
  - When you do not want to pass interface or abstract class dependence to internal class or other layer of the application
  - If the code refers to methods only, and does not need access to any other attributes or method of the class from which logic needs to be processed, delegates are better suited
  - Sometimes the event driven developement needs to be done

THE END

dr inż. Małgorzata Janik
malgorzata.janik@pw.edu.pl

Bonus Task

- **3.** Layers of the program:
- To separate function calls from user interface you should create **RestOfTheProgram** class, which will collect information about user choices and return the result (also, this way you can see how to pass delegates as parameters).
  - Class will consist of static methods, which will be called when 'Go!' button is pressed.
  - You should create one such method (ShowResult), and move the invokation of the delegate there

    → In short: just copy content of the Go! button click into separate class & function, and pass delegate as function parameter

- **Zmodyfikować** klasę **Output** dodając do niej składnik: **dwuwymiarową tablicę obiektów double.**
  - Tablica powinna mieć rozmiar 2 x 10 i powinna przechowywać wyniki obliczeń.
  - W pierwszym wierszu powinna przechowywać wyniki przekazywane do metody ShowDialogBox, a w drugim wyniki przekazywane do metody SaveToFile.
  - Kolejne kolumny przechowują kolejne wyniki.
  - Jeśli dodanie kolejnego wyniku wykraczałoby poza zakres (10 liczb) należy zresetować licznik i dodawać (nadpisywać) nowe elementy począwszy od indeksu 0.