# Assignment 1: RPN Calulator
# Object-Orientation, UML, Exceptions, I/O

Object-Oriented Program Development (TOCK13) Autumn 2025

## Contents

# 1 Introduction

## 1.1 Purpose

The assignment covers fundamental concepts in object-oriented programming (abstraction, polymorphism, inheritance, encapsulation, attributes/properties, methods, classes/objects, interfaces) and generic collections, exceptions, I/O, and delegates in C#, including UML (class diagrams).

The assignment is done in groups of three students.

## 1.2 Intended Learning Outcomes (ILOs

This assignment examines *parts of* the below intended learning outcomes (in teal).

**Knowledge and Understanding**

1. Show familiarity with fundamental modeling techniques in UML.
2. Display knowledge of fundamental techniques and terminology used in object-oriented programming.
3. Display knowledge of design principles in SOLID.
4. Display knowledge of design patterns in GoF and how SOLID relate to them.

**Skills and Abilities**

1. Demonstrate the ability to communicate a program design in a modeling language.
2. Demonstrate the ability to apply SOLID design principles, and GoF design patterns when developing an object-oriented application.
3. Demonstrate the ability to create object-oriented applications according to good object-oriented principles.

**Judgement and Approach**

1. Demonstrate the ability to assess the suitability of object-oriented program constructions with respect to a given problem.

## 2 Task

A customer has asked you to create a simple ***RPN Calculator***, where *RPN* is an acronym for *Reverse Polish Notation* (or *postfix notation*).

https://en.wikipedia.org/wiki/Reverse_Polish_notation
https://mathworld.wolfram.com/ReversePolishNotation.html
https://medium.com/@stevenpcurtis.sc/infix-postfix-prefix-and-reverse-polish-notation-299affa57acf

What is an RPN Calculator?
In maths, we are accustomed to use arithmetic expressions with *infix* notation. For example, we can express the sum of two numbers, A and B, with infix notation as A + B. In this case, *plus* is the operator +, *fixed inside the expression* between its two operands A and B. We therefore call this notation *infix*. Similarly, we can express an equivalent mathematical expression for the sum of two numbers with *prefix* or *postfix* notation. When using *prefix* notation, the operator is placed before its two operands + A B, whereas when using *postfix* notation the operator is placed after its two operands A B +. The *prefix* notation is also called *Polish Notation (PN)*, whereas the *postfix* notation is also called *Reverse Polish Notation (RPN)*. **Your calculator will use the *postfix* notation**.

According to the links above, you can use a **stack** as the data structure to simplify the processing when evaluating a *postfix* expression. Therefore, you will also use a **stack**, but with a *slightly different algorithm* so that you can better focus on **object-oriented programming**, which is the main purpose of this assignment.

You should process **postfix expressions** using a **stack** in the following manner:

1. The expression 3 4 5 * - is processed from left to right, i.e. the 3 first and the - last.
2. If the string is *split* using white space as the separator, the result is the following set of symbols (tokens): (3, 4, 5, *, -). These tokens are *pushed* on a stack in the same order they are processed (left to right):

   
   

   Here, 3 is at the bottom of the stack and - is at the top of the stack. **Note that you should *push* <u>all</u> symbols (tokens) in a RPN string to the stack before processing them**.
3. Then, the elements are *popped* from the stack, one by one, which calculations are made. When the stack is **empty**, the **correct answer has been calculated**.
   - pop - (this binary operator needs two operands before anything is calculated)
   - pop * (this binary operator needs two operands before anything is calculated)
   - pop 5 (this is the * operator's second/right operand)
   - pop 4 (this is the * operator's first/left operand)
   - Now the * operator has both its operands so the following is calculated: (4 * 5) = 20 (if we replace 4 5 * in the original expression, i.e. 3 4 5 * -, with 20 we get 3 20 –)
   - pop 3 (this is the - operators first/left operand)
   - Now the - operator has both its operands (20 is the - operator's second/right operand) so the following is calculated (3 - 20) = -17
   - Since the stack is **empty**, -17 is returned as the final result

# 3 The Customer's Specification of the Application

The application's main class should be called `Calculator` (i.e. not `Program`, so rename this) and should accept either 0 or 2 (*command line arguments*). If the application is started with 0 arguments, the user should be presented with the message **Enter an RPN expression <return> (empty string = exit):** via the standard output stream (`Console.Out`, e.g. via `Console.WriteLine()` or `Console.Write()`), followed by the user entering an RPN expression and pressing the `<return>` key. If the expression is valid, the result should be presented to the user via `Console.Out` in the format: **Result: x** where **x** is the answer. If any error occurs during the calculation, an error message should be displayed instead. In any case, the user is once again presented with the message **Enter an RPN expression <return> (empty string = exit):**. This loop of entering of RPN expressions and calculating the result continues until the user enters an empty string as an RPN expression, which causes the application to terminate with the message **The user exited the application**.

A possible session could look as below:

```
Enter an RPN expression <return> (empty string = exit):  3 4 5 * -
Result:  -17.0
Enter an RPN expression <return> (empty string = exit):  3 4 5 w -
InvalidTokenException:  w
Enter an RPN expression <return> (empty string = exit):  3 4 +
Result:  7.0
Enter an RPN expression <return> (empty string = exit):
The user exited the application
```

In the session above, the application `Calculator.exe` was started without any command line arguments. Then, the user entered the expression 3 4 5 * -, which was calculated as $3 - (4 * 5) = 3 - 20 = -17$ where the result was displayed to the user. The next input 3 4 5 w - contained an invalid character w, which causes the exception `InvalidTokenException` to be thrown. Then, another valid expression 3 4 + was entered, and calculated as $3 + 4 = 7$. Finally, the user entered an empty string (i.e. pressed <return> without entering a string), which causes the application to terminate with the message `The user exited the application`. If the application is started with 2 valid command line arguments according to `Calculator.exe input.txt output.txt`, RPN expressions should be read, row by row, from the file `input.txt` and the output should be written to the file `output.txt`.

The example below shows the contents of a source file (`input.txt`) and its destination file (`output.txt`):

| | input.txt (source) | output.txt (destination) |
|---|---|---|
| 01 | 3 4 + | 7.00 |
| 02 | 3 4 - | -1.00 |
| 03 | 4 3 - | 1.00 |
| 04 | 3 4 * | 12.00 |
| 05 | 3 4 / | 0.75 |
| 06 | 4 3 / | 1.33 |
| 07 | 3 4 % | 3.00 |
| 08 | 4 3 % | 1.00 |
| 09 | 34 56 + | 90.00 |
| 10 | 3 4 5 * + | 23.00 |
| 11 | 3 4 5 * - | -17.00 |
| 12 | 3 4 + 5 * | 35.00 |
| 13 | 3 4 - 5 * | -5.00 |
| 14 | 3 4 + 5 6 + * | 77.00 |
| 15 | 3 4 5 + * 2 * | 54.00 |
| 16 | 3 4.8 5.7 + * 2.5 * | 78.75 |
| 17 | 100 2 4 / 5 6 + * - | 94.50 |
| 18 | 239 100 4 2 / 5 6 + * - % | 5.00 |
| 19 | 4 0 / | DivideByZeroException: 4.00/0.00 |
| 20 | 4 0 % | DivideByZeroException: 4.00/0.00 |
| 21 | 4 / 3 | InvalidOperationException |
| 22 | A + B | InvalidTokenException: A |
| 23 | 5 & 6 | InvalidTokenException: & |
| 24 | 1 2 + | 3.00 |

On lines 19 and 20, the exeption `DivideByZeroException` has been thrown since the program tried to divide four by zero (4 / 0), and tried to calculate 4 modulus 0 (4 % 0), respectively. On row 21, the exception `InvalidOperationException` has been thrown since the user entered an expression in infix notation (4 / 3) instead of postfix notation (4 3 /). On rows 22 and 23, the exception `InvalidTokenException` has been thrown since e.g. A isn't a valid operand (i.e. not a number) and & isn't a supported arithmetic operator (+ - * / %).

At least the following arithmetic operators should be supported by the application:

- + : the sum of two operands `A` and `B`, e.g. `A B +`
- - : the difference of two operands `A` and `B`, e.g. `A B -`
- * : the product pf two operands `A` and `B`, e.g. `A B *`
- / : the quotient of two operands `A` and `B`, e.g. `A B /`
- % : the modulus of two operands `A` and `B`, e.g. `A B %`

Both integers and floating point numbers (positive and negative) should be supported as operands.

If the application is started with anything but 0 or 2 command line arguments, the user should be presented with message **Syntax: Calculator [source destination]** via the standard output stream (`Console.Out`), followed by terminating the application.

# 4 Architecture

The customer wants you to use an **MVC** (*Model View Controller*) architecture (https://en.wikipedia.org/wiki/Model-view-controller).

In the **MVC** architecture, all C# classes are separated into three main parts:

- **Model**: classes associated with the domain (logic and data), which in this case is the actual calculator with its tokens, stack, calculations, etc.
- **View**: classes associated with the user interface (user input and output/presentation).
- **Controller**: classes that drive the application, control specific use cases, and that separate the Model classes from the View classes.

Since we haven't covered software architecture in the course, this means the following for you:

- Create a new Visual Studio project
  1. When Visual Studio starts, choose **Create a new project** (alternatively, if Visual Studio is already open, choose **File** → **New** → **Project** from Visual Studio's main menu).
  2. Select the project type **Console App** (for C#).
  3. Name the project **Calculator** (in the `Project name` field at the top of the window).
  4. Choose the (*framework*) to **.NET 8.0 (Long-term support)**, and checked the checkbox **Do not use top-level statements**.
  5. Click the **Create** button.
- Create 3 folders in the Visual Studio project.
  1. In *Solution Explorer* (in Visual Studio's right margin), right-click the project node (with the name **Calculator**) and choose: **Add** → **New Folder**.
  2. Name the folder **Model**.
  3. Repeat steps 1-2 but name the folder **View**.
  4. Repeat steps 1-2 but name the folder **Controller**.
- Program structure and (*command line arguments*)
  1. Right-click the file `Program.cs` in *Solution Explorer*, choose **Rename**, and name the file `Calculator.cs` (press **Yes** in the pop-up window).
  2. Double-click the file `Calculator.cs` in *Solution Explorer* which opens the source code file in the editor.
  3. Notice the following structure in the source code file:
     - At the top of the file, there are usually a number of `using` directives that import *namespaces* into the source code file (e.g. `System` that contains the type `Console`). In the current version of Visual Studio/C# something called **global implicit using directives** is used, why you won't initially see any `using` directives. This means that Visual Studio automatically (implicitly) adds `using` directives for the following namespaces in each source code file for a project of type **Console App**; `System`, `System.Collections.Generic`, `System.IO`, `System.Linq`, `System.Net.Http`, `System.Threading`, and `System.Threading.Tasks`. Although, if you use types from other namespaces in a source code file, you must add the namespaces yourself with `using` directives at the top of the source code file.

By importing a `namespace` with the `using` directive, you can use all types (contained in the namespace) in the source code file without having to write the **fully qualified name** of the type. The **fully qualified name** of a type is the name of the type, preceded by all namespaces the typ is nested within. For example, the **fully qualified name** of the `Console` type is `System.Console` since it is defined in namespace `System`, and the **fully qualified name** for the generic type `List<T>` is `System.Collections.Generic.List<T>` since it is defined in namespace `Generic` which, in turn, is in namespace `Collections` which, in turn, is in namespace `System`.

- After the `using` directives, the **namespace** `Calculator` is defined as the application's top namespace.
- In **namespace** `Calculator`, the **class** `Calculator` (the class that was renamed from `Program`) is defined, which constitutes the application class containing the program's `Main()` method.
- The `Main()` method has a parameter `args` of type *array of strings*, i.e. `String[]`, which contains any *command line arguments* passed to the program when it is started.

4. Right-click the project node **Calculator** in *Solution Explorer* and choose: **Properties**.
5. Make sure the **Application** tab is selected (to the left) in the window that pops up.
6. Here, you can e.g. set the name of the assembly, what framework version to use, type of application (e.g. **Console App**, **Class Library**) and the application's **default namespace** that has the value **$(MSBuildProjectName.Replace(" ", "_"))**, which is a **macro** that results in the project name (**Calculator**) being used as the **default namespace**. The **default namespace** is the **namespace** Visual Studio will automatically create for new source code files added to the project.
7. Choose the **Debug** tab further down in the left part of the window.
8. To the right in the window, (under *Debug* → *General*) there is a link with the text **Open debug launch profiles UI**. If you click this link, another window is opened, where command line arguments can be set (in the field *command line arguments*) when debugging the program in Visual Studio. These command line arguments are entered into the text box as words separated by a space, which will be available via the `Main()` method's `String[] args` parameter. Close the window by clicking on the $x$ button in the top right of the window, and also close the **properties** window by clicking its $x$ button.
9. Right-click the folder **Controller** under the project node in Visual Studio, choose **Add** → **New Item** and select **Class** under **Installed** → **Visual C# Items** (alternatively, choose **Add** → **Class** which does the same thing). Name the new source code file `CalculatorController.cs` and click the **Add** button.
10. Notice that the new source code file `CalculatorController.cs` was placed in the **Controller** folder in *Solution Explorer*, and that the course code file that was opened in the editor has namespace `Calculator.Controller`. The first part of the namespace name, i.e. `Calculator`, is the program's **default namespace** (which could be configured in the project's **properties** in step 6 above). The second part of the namespace name, i.e. `Controller`, is the name of the folder the source code file was created in (in *Solution Explorer*). I.e., the class `CalculatorController` is in namespace `Calculator.Controller`.
11. By using folders, and creating/organizing new source code files in them, a good logical program structure is obtained (i.e. the types are organized logically into nested namespaces in the assembly).

- The following program structure should be used in the assignment:

    1. **Namespace** `Calculator` should contain the **application class** `Calculator` (which was renamed from `Program`) which, in turn, contains the application's `Main()` method.
    2. Namespace `Calculator.Model` should contain all C# classes associated with the domain, i.e. that have to do with the calculator's logic and data (calculations, tokens, the stack, etc.).
    3. Namespace `Calculator.View` should contain all C# classes associated with the user interface, i.e. input of RPN strings from the user, and presentation of results and error messages to the user.
    4. Namespace `Calculator.Controller` should contain a class `CalculatorController` that drives the actual application. Furthermore, this class ensures that the **model**-classes are separated from the **view**-classes, i.e. no **model**-class should know about (have an association to) any **view**-class, and no **view**-class should know about (have an association to) a **model**-class. Only the `CalculatorController` class should know about **model**-classes and **view**-classes.
    5. You can, of course, extend this structure, e.g. by defining a namespace `Calculator.Exceptions` for user-defined exception classes or a namespace `Calculator.Model.Tokens` for operator and operand classes, which are nested under namespace `Calculator.Model`.

- The **class** `Calculator` (that was renamed from `Program`) in namespace `Calculator` should create an instance of the class `CalculatorController`, and call a method named e.g. `Run()`.

- The `CalculatorController` class in namespace `Calculator.Controller` should contain a method named e.g. `Run()`, that iterates through the loop below until either an empty string is entered by the user, or all rows in the source (input) file have been processed:

    1. Loop
    2. Get the next RPN row (from the user of the source/input file)
    3. If the RPN row is null or an empty string
    4. Exit
    5. Else
    6. Ask the calculator to compute the result of the RPN expression
    7. Present the result (to the user or the destination (output) file)

Thus, the `CalculatorController` class drives the application and creates a separation between the **model**-classes and the **view**-classes (this increases e.g. the maintainability of the code).

The `CalculatorController` class gets its input from a class in the user interface, i.e. a **view**-class, and sends the input to the RPN calculator, that should accept an RPN expression in the form of a string, split this to `Tokens` (operators and operands) and *push* these on the **stack**. To calculate the result, these are the *popped* from the **stack** as the arithmetic calculations are made. The result is then returned to the `CalculatorController` class which, in turn, asks a **view**-class to present the result. Note that your **stack** should only know about `Tokens`! A `Token` can be either an `Operator` or an `Operand`, where an `Operand` can contain a number. An `Operator` can be either a `SumOperator`, `SubtractOperator`, `MultiplyOperator`, `DivideOperator` or `ModulusOperator` (all operators are binary, i.e. operate on two operands). Create an object-oriented design for this, where a certain operator knows how to calculate the result for its two operands. Use a *generic collection* class for your **stack**. Furthermore, the customer wants to be able to replace the underlying implementation for the **stack** in a future version of the application, without having to change its public interface, and without having to make changes to multiple files. Take this into consideration when designing your application.

# 5 Requirements Specification

Design documentation (i.e. a design model consisting of UML class diagrams + documentation comments in the source code files) for all types (classes, interfaces, etc.) should be handed in together with your code. Start by thinking about what classes/interfaces should be part of the solution and specify the classes/interfaces, attributes/properties, methods, and relationships (associations, etc.) between classes/interfaces in UML. Thinking through a design for an application before actually coding the application often results in a well-structured application.

1. The `Main()` method should be in the class `Calculator.Calculator`. This class should create an instance of the `Calculator.Controller.CalculatorController`.
2. The program should be a console application, i.e. of project type **Console App** (.NET 8), organized into classes/interfaces. Each class/interface should be defined in its own source code file (i.e. do not define multiple classes/interfaces in the same course code file). Encapsulation, Inheritance and Polymorphism should be used.
3. When objects (instances of classes) are created, all necessary information should be passed in as arguments to the constructor (i.e. don't call a parameter-less constructor, followed by assigning values to public properties, but instead pass the values as arguments and assign them to attributes in the constructor's body).
4. Each operator and operand should have an overridden `ToString()` method that returns a string representing the state of the object. The `ToString()` method is defined as `virtual` in `System.Object`, so (`override`) the method in the subclasses. All classes in .NET inherit from `System.Object`.
5. The program should include exception handling so that it doesn't crash when errors occur.
6. Use user-defined `Exception`-classes as necessary, but at least one user-defined exception (that contains at least one user-defined attribute/property) should be included. Note! You might get some name clashes between your user-defined exceptions and the standard library's exception classes. If this occurs, use the following syntax when importing your own exception classes in your source code files:

   ```
   using DivideByZeroException = Calculator.Exceptions.DivideByZeroException;
   ```

7. All `Tokens` for one RPN expression should be pushed on the stack before popping them and performing calculations with them.
8. The two user interfaces (one interactive, console-based and one non-interactive file-based) should be supported.
9. At least the five binary operators (+ - * / %) should be supported.
10. The MVC architecture should be followed and `CalculatorController` should contain a `Run()` loop, as described above, to drive the application.
11. The implementation of the stack should easily be replaced in a future version of the application, without having to modify its public interface, and without having to modify multiple source code files.

# 6 Other Information

Work in groups of 3, and choose your group on Canvas.

## 6.1 Submission

The assignment is submitted via Canvas (via the assignment's Canvas page), where all files are handed in as one archieve file(.zip or .rar). The following should be included in the archive:

- **Design Documentation**: (Drawio or Visio) UML class diagrams (design model) including classes/interfaces, attributes/properties with data types and visibility, methods with signatures and visibility, relationships between types (inheritance, associations), etc., where static and abstract types and members, and multiplicity, is shown using the correct UML notation.
- **Source Code**: Zip the whole **Solution** folder with sub-folders (you can remove the sub-folders **bin** and **obj** under each **Project** folder, if you like). Document relevant types and members with comments in your code (see Appendix A). Use C#'s naming- and coding conventions (see Appendix B).

There are 3 submission and examination opportunities for the assignment according to the schedule (the information is also on the Canvas page for the assignment). The following applies for each submission opportunity:

- Deadline 1: Friday, 3 October 2025, 23:59 (Week 40)
- Deadline 2: Friday, 19 December 2025, 23:59 (Week 51)
- Deadline 3: Friday, 6 March 2026, 23:59 (Week 10)

## 6.2 Grading

This assignment is graded either as Failed (U) or Passed (G).

# A Appendix: C# Documentation Comments

https://docs.microsoft.com/en-us/dotnet/csharp/codedoc

To document types (e.g. classes) and methods, the msot common documentation tags are `<summary>`, `<param>` and `<returns>`.

To document a class, the documentation tags are written above the class name, as below:

```
/// <summary>
/// Describe the class here
/// </summary>
```

To document a method, the documentation tags are written above the class name, as below:

```
/// <summary>
/// Describe the method here
/// </summary>
/// <param name="paramname"> Describe parameter here </param>
/// <returns> Describe return value here </returns>
```

To document attributes and properties, the same syntax can be used as for a class, but isn't as common.

# B Appendix: C#'s Naming- and Coding Conventions

Microsoft has defined a number of naming conventions for C#. https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

Microsoft has also defined a number of coding conventions for C#. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

We follow the conventions starting with assignment 1, so that you produce C# code in just the same way as other C#/.NET developers do.

# C Appendix: Code Snippets

https://docs.microsoft.com/en-us/visualstudio/ide/visual-csharp-code-snippets?view=vs-2022

### Programming Idioms

In Visual Studio, there is a quick way to produce code skeletons for common programming idioms via **code snippets**. For example, if you enter `cw` followed by a `tab`, i.e. `cw <tab>`, then Visual Studio replaces this with `Console.WriteLine()`. Similarly, there are code snippets for:

```
cw → Console.WriteLine()
prop → public int MyProperty { get; set: }
ctor → public <classnam> { }
```

### Override

When implementing a class that inherits virtual or abstract methods, you can write `override <space>` which pops up a list of methods that can be overridden, and when selecting one, Visual Studio generates a method skeleton for you.

### Implementing Interfaces

When implementing an interface in a type, place the cursor directly after the interface name and press `<ctrl> + <.>`, which will give you the choice to auto-generate method skeletons for all the method prototypes in the interface.

### Using

To automatically add a using directive for a type in a source code file, place the cursor somewhere in the type's name and press `<ctrl> + <.>`, which will give you a choice to automatically add the using directive for you (as long as the project has a reference to the assembly containing the type).

### Documentation Comments

To generate a skeleton for documenting e.g. a class or method, place the cursor just above the name and enter three lash characters `///`.

### Region

To hide/show code in the editor, you can click on the +/− symbols in the left margin. You can also create your own named **regions** that can be hidden/shown be entering the code between #region and #endregion directives, e.g.:

```
#region Attributes
// Define attributes here
#endregion

#region Properties
// Define properties here
#endregion

#region Constructors
// Define constructors here
#endregion
```

```
#region Methods
// Define methods here
#endregion
```