TODO LIST

Task 1 - Overview

This application is a simple manager of TODOs.
Each TODO is simply a description of a thing to be done,
for example "Order a birthday cake for the upcomming birthday".

Conventions:
- Each description must be unique
- TODOs can be added, removed, or listed
- Must be a Console-App

Main application workflow

When the application starts, it shall print:

A:

```
Hello!
What do you want to do?
[S]ee all TODOs
[A]dd a TODO
[R]emove a TODO
[E]xit
```

Description:

The User must select one of the given **S, A, R, E** options. The selected option can be both uppercase or lowercase, so it doesn´t matter if the user types "S" or "s" - both should go to the "see all todos" case. The user can keep selecting different options until the option "[E]xit" is selected, which will close the application.

"A:" = Ausgabe, "E:" = Eingabe

TODO LIST

Task 2 - Input Handling

It´s getting a little tricky now ;) Our goal is to implement key reading as well as check for valid input

Main application workflow

Scenario	User action	Result
Sunny day	User selects any of the S,s,A,a,R,r,E,e options.	The correct option is handled. After it is finished, the choice of options is printed again (unless the **Exit** option was chosen, which closes the app).
Incorrect or empty input	User does not select any option (empty choice), or the selected option is not valid.	“Incorrect input” is printed to the console. Then, the selection of choices is printed again. It is repeated until the user provides the correct input.

TODO LIST

Task 3 - Option "S" - See all TODOs

This should be easy: In this step we just need the app to print out all the TODOs. We also want a prefix number, starting with 1 for the first TODO:

1. Order a cake for the birthday party.
2. Buy train tickets for the weekend.
3. Take Lucky to the vet.

Afterwards,the main menu should be printed again to wait for the next user input.

Main application workflow

Scenario	User action	Result
Sunny day		The list of all TODOs is printed to the console. Then, the choice of options is printed again.
There are no TODOs at all		“No TODOs have been added yet.” is printed to the console. Then, the choice of options is printed again.

✓ TODO LIST

Task 4 - Option "A" - Add a TODO

After selecting this option, the application shall print:

```
Enter the TODO description:
```

Then, the user must enter a unique, non-empty description of a TODO. Once it is   done, the application prints:

```
TODO successfully added: [DESCRIPTION ]
```

Where the DESCRIPTION should be the TODO description the user provided. A TODO shall be added to the collection of TODOs (it can be seen by selecting the “S - See all TODOs” option). A ter a TODOs is added, the application should print again “What do you want to do?” with all available options.

Main application workflow

Scenario	User action	Result
Sunny day	User inputs a unique, non-empty description of a TODO.	“TODO successfully added: [DESCRIPTION]” is printed to the console. TODO is added to the collection. The choice of options is printed again to the console.
Empty description	The description the user provided is empty.	“The description cannot be empty.” is printed to the console.No TODO is added. “Enter the TODO description” is printed again.
Non-unique description	There is already a TODO with the same description asthe user provided.	“The description must be unique.” is printed to the console.No TODO is added. “Enter the TODO description” is printed again

TODO LIST

Task 5 - Option "R" - Remove a TODO

After selecting this option, the app should ask:

Select the index of the TODO youwant to remove:

Then, the list of all TODOs shall be printed, similarly as described for the “S - See all TODOs” option.

The user should select a correct index from the given list (please notice that this list is indexed from 1, so when the user selects“1,” the first TODO from this list should be removed). After the index is selected, the TODO should be removed,which means it will no longer be shown when the S - See all TODOs option is selected. Then, the following message is printed:

TODO removed: [DESCRIPTION]

…where the DESCRIPTION is the description of the TODO that has just been removed. Then the application should return to its usual state.

Main application workflow

Scenario	User action	Result
Sunny day	User inputs a correct index of a TODO.	“TODO removed: [DESCRIPTION]” is printed to the console. TODO is removed from the collection of TODOs. The choice of optionsis printed again to the console.
There are no TODOs		Same as step 3.
Empty index	The user input is empty.	“Selected index cannot be empty.” is printed to the console.No TODO is removed. “Select the index of the TODO youwant to remove” is printed again.
Invalid index	The index the user inputs is either not a number or it is not a valid index in the collection.	“The given index is not valid.” is printed to the console.No TODO is removed. “Select the index of the TODO youwant to remove” is printed again.

TODO LIST

Task 6 - Option "E" - Exit

Now all you have left to do is find a way to close the application. This wraps up the main funktionality of our TODO-App :)

✓ TODO LIST

Task 7 - Refactoring

Depending on your previous experiences in coding, you might have taken several approaches to solving the previous tasks. In this step, consider the following ways to optimize your code:

If your code ended up as one big block, you might want to break each step up into its own function.

Compare your solutions for steps 3 and 5 and see if you can reuse some code in a function.

If your code so far is just floating outside of any scope, you might have to enclose it into a namespace for the functions to work.

How did you handle your conditionals? Consider how many steps your processor has to do to arrive at the branch it needs to take. Is there a way to simplify your scopes? Are else-branches an option, or a switch case?

Basic Syntax - Functions/Methods

To declare a function, you need a **return type**, your **function name** followed by parenthesis **()** (which may include **parameter declarations**), and your function body which is enclosed by curly braces **{}**.

Declaration:

```
int MyFunction(float parameter)
{
        int output;
        // some code here, you may do something with your parameter
        return output;
}
```

Use:

```
float input = 3.14;
int calculate = MyFunction(input);
```

In your code, the function call is replaced by its return value, which can be used as any other value. If your function does not need to produce a value, for example print text to the console, you may use the return type **void**. In this case, your function does not need the **return** instruction, and can't be used to assign a value. You can list as many parameters as you want, or none at all.

✓ TODO LIST

Task 8 - Object Oriented Programming

C# is an object oriented programming language - which means it supports, or rather endorses the use of classes and objects.

Without class structure, your programm so far might be a collection of free floating functions. Consider how these tasks relate to one another and how to encapsulate them into a common entity.

For the scope of this project, it should be enough to create a class that simply encompasses your main S,A,R,E options, as well as any other funtions that arose along the way.

Another consideration is making your code maintainable. So far, your TODOs simply consist of descriptions. What if you want to add a parameter that tells you if the task is completed? Maybe your want to add a due date? All these parameters belong to a TODO, so consider encapsulating them into their own class.

Basic Syntax - Classes

Classes are declared with the **class** specifier, followed by the **class name** and a scope defined by curly braces { }:

```
class MyClass
{
        public int number; // Attribute

        public void DoSomething(string argument) // Function
        {
                String += argument;
                Console.WriteLine(String);
        }
}
```

Classes may include attributes such as variables of any type, or functions. By default, those attributes are only within the class, so you need to first create an instance of the class, called an object:

```
MyClass object = new MyClass();
```

The **new** keyword is necessary to reserve memory for the object.

✓ TODO LIST

Access Modifiers

Per default, Classes won't let you access its attributes from outside the class. Access modifiers control how you may access a varibale or function. **private** (default) will only allow the class itself to access it. **public** lets external code (from outside the class' scope) access your attribute, and **protected** lets related classes access it (inheritance).

C# Access Modifiers (w3schools.com)

Constructor

C# can create an object for you without further specifications. If you need a default object to have a certain form, or if you want to fill out some of its attributes on creation, you can write your own **constructor**.

A **constructor** is a function without a return value and its name must be identical to the class name. It is called whenever you create an object with parameters matching your constructor funktion.

C# Constructors (w3schools.com)

Advanced Syntax - Classes

```
class MyClass
{
        public int number; // Attribute
        string Text {get; private set;} // Property, cannot be changed from
                                           outside the class

        public MyClass() // Constructor
        {
                number = 2;
        }

        public MyClass(int number, string text)  // overloaded Constructor
        {
                number = number;
                Text = text;
        }
}
```

✓ TODO LIST

Properties and attributes in the class

Classes in C# can store their own data. All data belonging to an object of the class is stored in Attributes and Properties.

You can easily manipulate data in those structures, retrieve or change information. Attributes have better performance compared to properties because they require storing copies of your data in a separate variable and have to call additional functions - your getters and setter. You may also define default values or **access modifier** for your attributes.

Basic Syntax - Properties / Attribute

Properties and Attributes are declared within the **class**, followed by the **data type,** its **name** and a scope defined by curly braces { }:

```
class MyClass
{
    public int number; // Attribute
    string Name {get; set;} = "My String"; // Property with default value
}
```

To use those data at first you have to create object of the class where information can be stored:

```
MyClass object = new MyClass();
```

Now you can store information within the object. Without using the default value or defining the value in functions, the value will be **null**, which represents nothing. To get the value of your property you may do so by calling your object, followed by a dot **.** and the attribute name:

```
var myNumber = object.number
```

The assignment of a value to an attribute is done similarly:

```
object.number = 2;
```

With the help of your setter and getter, you may manipulate or valudate your value before saving or retrieving it.

C# Properties (Get and Set) (w3schools.com)

TODO LIST

Task 9 - Data handling

So far, the data of your TODO app will get lost every time your close and restart it - consider a way of storing it.

The simplest way would to to save them to a text file and reading it back on restart.

Consider your data format, will a simple csv suffice, or maybe you want to structure your data as an xml or json file?

You can also create an SQL database. Consider the app functionality that communicates with the database.

For SQL database you can use the SQLConnection class, or Entity Framework ().

Use .NET to connect and query a database on Windows, Linux, or macOS - Azure SQL Database & SQL Managed Instance | Microsoft Learn

Getting Started - EF Core | Microsoft Learn

TODO LIST

Task 10 - User interface

If you're still bored, consider creating a graphic user interface. So far, we only have a text-based console application. It does the job, but it's not very appealing to the end user.

Search for a suitable framework to create a windowed application. One option for web applications is Razor Pages.

Introduction to Razor Pages in ASP.NET Core | Microsoft Learn