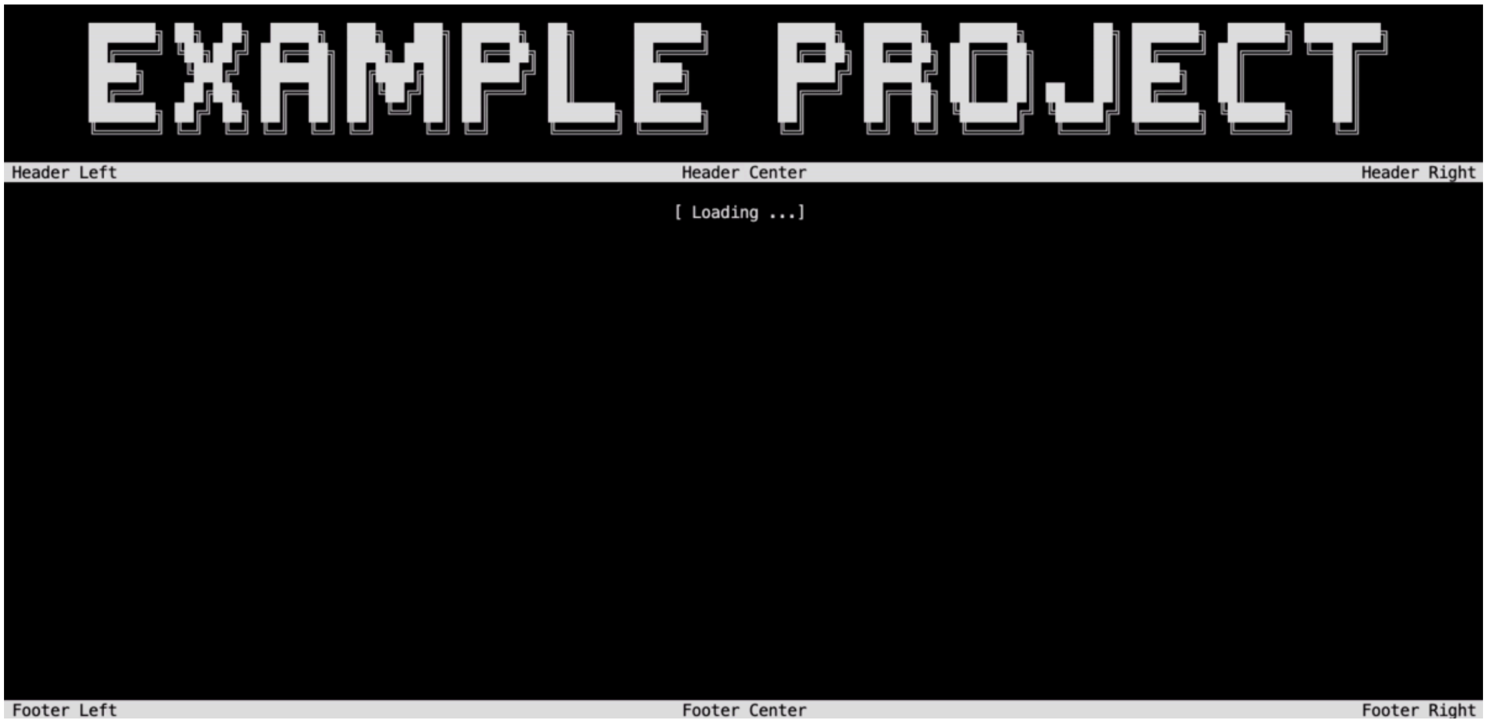
# Welcome to the docs

version v3.6.0 downloads 11k stars 30 coverage 96% License GPL v2.0 Code lines 29k



```
EXAMPLE PROJECT
Header Left                         Header Center                         Header Right
                                    [ Loading ...]

Footer Left                         Footer Center                         Footer Right
```

## Table of contents

Welcome to the documentation of the `ConsoleAppVisuals` library. This is a simple and easy-to-use library that allows you to create visual elements in the console. Here are all the resources available:

- Introduction: find the basic concepts and the first steps into the library to learn how to use it.
- Elements: find all the visual elements available in the library with their description.
- References: find all methods, properties and classes with their description and all arguments available.
- Examples: find some examples to learn how to use the library in concrete use cases.
- Articles: find some additional library-related articles.
- Legacy: find the outdated documentation of the library for the versions 2.x.x and below.

## Roadmap

The library is still in active development. The next feature and bug resolutions are listed in the Project section of the GitHub repository.

## Supported .NET versions

| Version | Supported |
|---|---|
| net9.x | ✅ |
| net8.x | ✅ |
| net7.x | ✅ |
| net6.x | ✅ |
| < net6.x | ❌ |

## Security Policy

Consider reading our SECURITY policy to know more about how we handle security issues and how to report them. You will also find the stable versions of the project.

## Acknowledgments

Consider reading the ACKNOWLEDGMENTS file. It's a testament to the collaborative effort that has gone into improving and refining our library. We're deeply grateful to all our contributors for their invaluable input and the significant difference they've made to the project.

It also lists the open source projects that have been used to build this library until now.

## Contributing

Contributions are what make the open source community such an amazing place to learn, inspire, and create. Any contributions you make are **greatly appreciated**. To do so, follow the steps described in the CONTRIBUTING file.

We are always open for feedback and discussions. If you are using our library and want to share your use case, or if you have any suggestions for improvement, please feel free to open an issue or open a discussion on our GitHub repository. Your input helps us understand possible use cases and make necessary improvements.

Do not hesitate to **star** and **share** the project if you like it!

# Basic concepts

This section is made for you to understand what is **ConsoleAppVisuals**, its purpose and use flow. We will also guide you into the creation of your first project until the advanced use of the library with data visualization and menus management.

## What is ConsoleAppVisuals?

The ambition of ConsoleAppVisuals is to provide the best compromise between an **easy-to-use library** and a **complex tool** to create console applications with visual elements. The library is designed to be simple to use and to provide a wide range of visual elements to make your console application more stylish and useful.

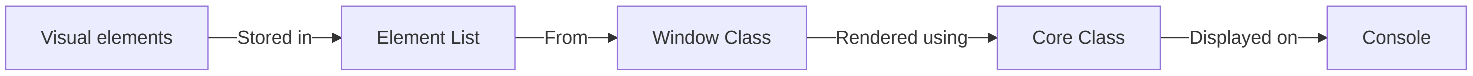
## Working principle

It is relies on the concept of "visuals" which are elements that can be displayed in the console. There are two types of visuals:

- **Passive visuals**: elements that do not provide any interaction, you may display several from the same type at the same time
- **Interactive visuals**: elements that provide an explicit interaction and create a response that can be collected, you may display only one at a time
- **Animated visuals**: in-between passive and interactive visuals, they provide an interaction to stop them but do not require a response, you may display only one at a time.

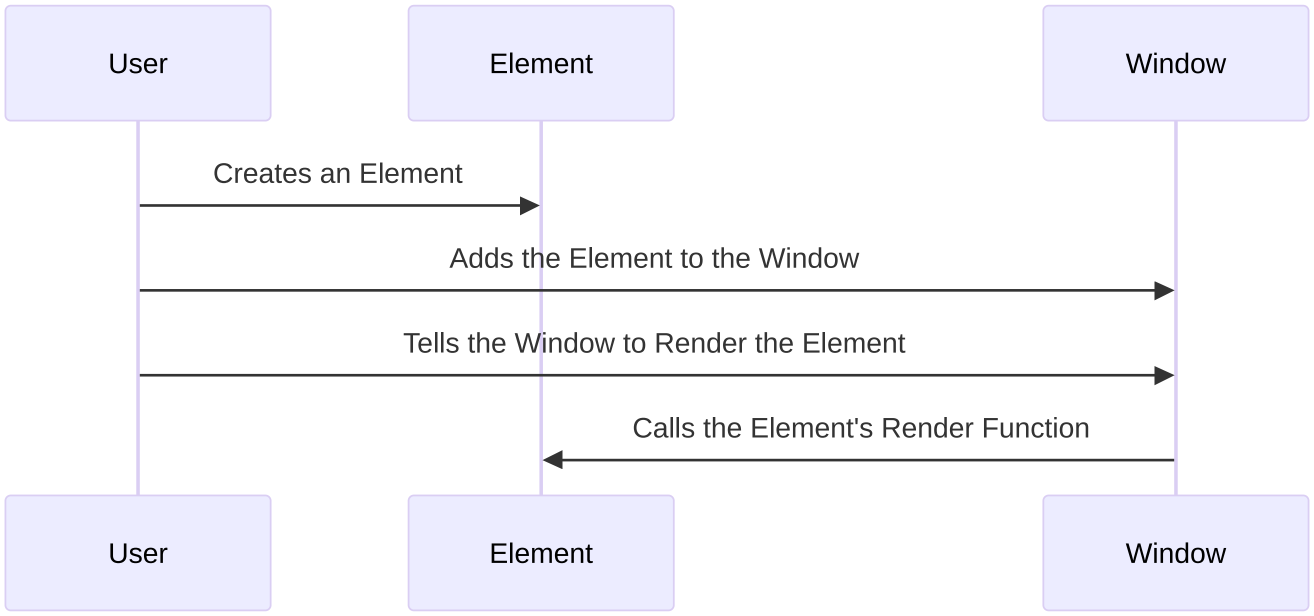
These visuals are stored in `Window` as a list. From this class, you can display, add, remove, or update the visuals. Each one of the visual element has its rendering method that will be called from the `Window` class.

The basics of the interaction between the library and the console are defined in the `Core` class.



## Use flow

When you want to create an element and display it, here is the basic visualization of the use flow of the library:


User
Element
Window
Creates an Element
Adds the Element to the Window
Tells the Window to Render the Element
Calls the Element's Render Function
User
Element
Window

In C# terms, the use flow can be interpreted like this:

1. Creating an element:

```
Title exampleTitle = new Title("Hello world!");
```

2. Adding it to the `Window`:

```
Window.AddElement(exampleTitle);
```

3. Rendering the element:

```
Window.Render(exampleTitle);
```

## First steps

Now that you have the basic concepts, let's dive into this guided path to learn how to use the library:

1. Create a simple console application
2. Explore element options
3. Discover data visualization
4. Manage multiple menus

---

Have a question, give a feedback or found a bug? Feel free to open an issue or start a discussion on the GitHub repository.

# First app

This tutorial will show you how to create a simple console application using the `ConsoleAppVisuals` package. You will learn:

- How to add elements
- Discover: `Title`, `Header`, `Footer`, `FakeLoadingBar`, `Prompt` and finally `Dialog` elements
- How to get the response from the user
- How to exit the application

## Setup

First, let's create a dummy project to work with. Please choose your method according to your preference:

.NET CLI    Visual Studio

Open your terminal and navigate to the folder where you want to create your project. Run the following command:

```
dotnet new console --output MyApp --use-program-main
```

If your file structure is like this:

```
Example_project  <-- root
└───MyApp
    ├───obj
    ├───MyApp.csproj
    └───Program.cs
```

Jump into the `MyApp` folder:

```
cd MyApp
```

Finally, run the following command to install the library:

```
dotnet add package ConsoleAppVisuals
```

**TIP**

Consider running the same command to update the package to the latest version, stay tuned!

Open the `Program.cs` file and ensure that the content is the following:

```
namespace MyApp
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

.NET CLI Visual Studio

Let's try to run the app by typing the following command in your terminal:

```
dotnet run
```

## Building the app

**WARNING**

Add these using statements at the top of your `Program.cs` file to use the `ConsoleAppVisuals` package and its elements:

```
using ConsoleAppVisuals;
using ConsoleAppVisuals.PassiveElements;
using ConsoleAppVisuals.InteractiveElements;
using ConsoleAppVisuals.AnimatedElements;
```

Sometimes, you will need to add more statements. All available namespaces are available in the references section.

Let's start by removing `Console.WriteLine("Hello World!");` instruction and adding the following line to your `Main` method to set up the console (clear and set the cursor invisible):

```
Window.Open();
```

Now, let's create a minimal app with a `Title`, a `Header`, a `Footer`, a `FakeLoadingBar` and finally a `Prompt` element.

## `Title`

Now we can use all the elements from the package. Our first *passive* element will be a `Title`. Learn more

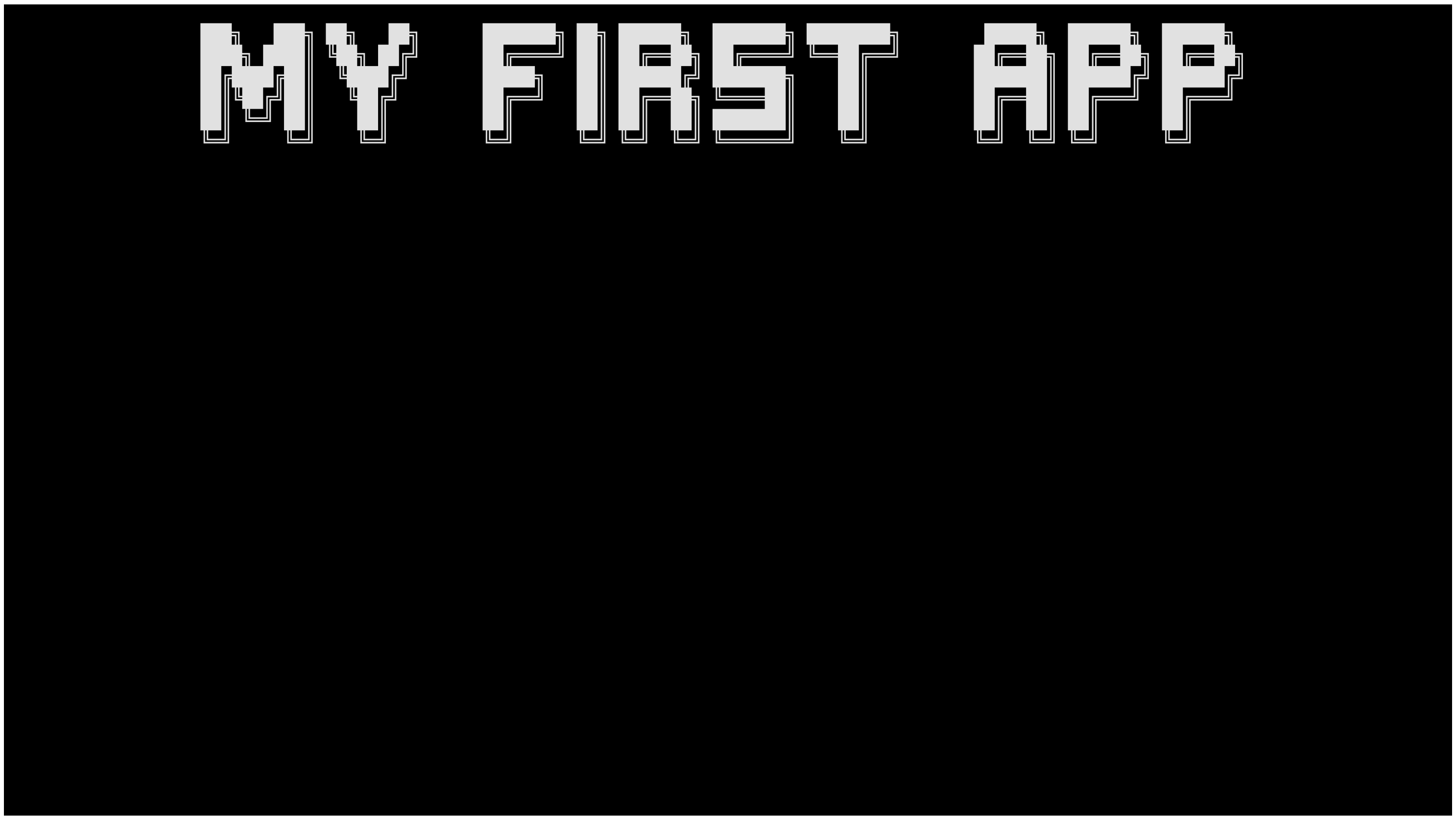
```
Title title = new Title("My first app");
```

Then we can add it to the `Window`:

```
Window.AddElement(title);
```

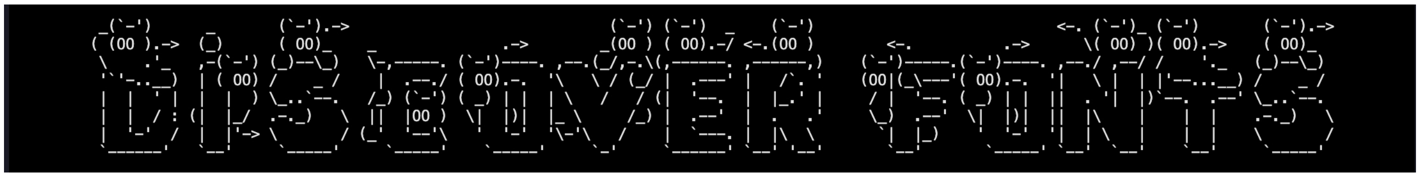
And finally, we can render the `Title` from the `Window`:

```
Window.Render(title);
```



**TIP**

You may update the style of the `Title` element like the one below by giving a look at this article: [create and use fonts](#).

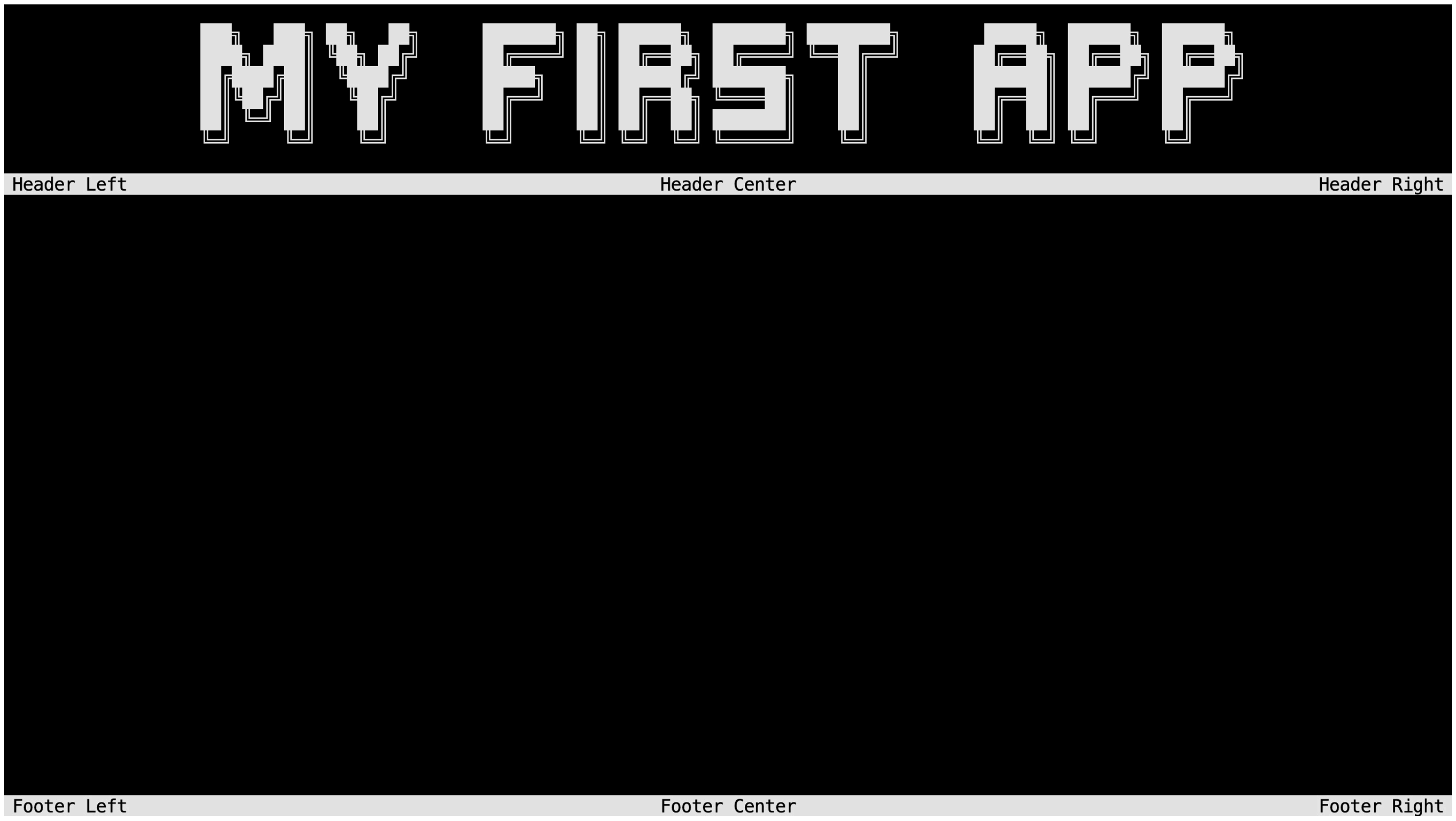


## `Header`, `Footer`

Add the [`Header`](#) and [`Footer`](#) *passive* elements to the `Window`:

```
Header header = new Header();
Footer footer = new Footer();
Window.AddElement(header, footer);

Window.Render();
```

Instead of rendering each element separately, we rendered the `Title`, `Header` and `Footer` elements at once using the `Window.Render()` method. So you may remove the `Window.Render(title)` instruction.



## FakeLoadingBar

Now let's add a `FakeLoadingBar` *animated* element to your previous code and run it. Learn more

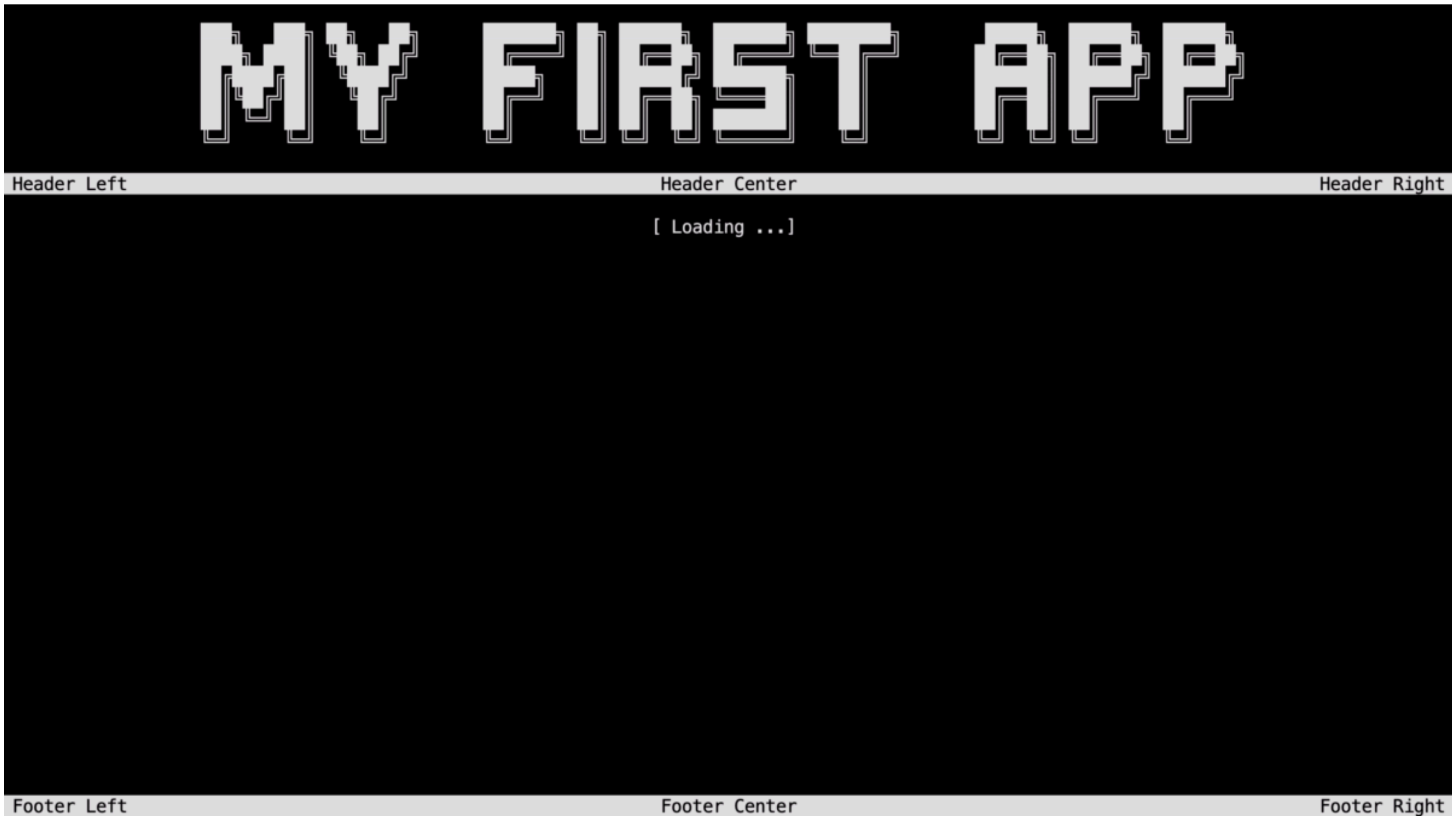
```
FakeLoadingBar loadingBar = new FakeLoadingBar();
Window.AddElement(loadingBar);

Window.Render(loadingBar);
```

**WARNING**

As you may have noticed, we have the same output as earlier. No loading bar was rendered on the console. **Passive elements** are **activated by default** when added to the window. On the contrary, **interactive and animated elements** need to be **activated manually**.

To do so, **replace** the `Window.Render(loadingBar)` instruction with the following:

```
Window.ActivateElement(loadingBar);
```

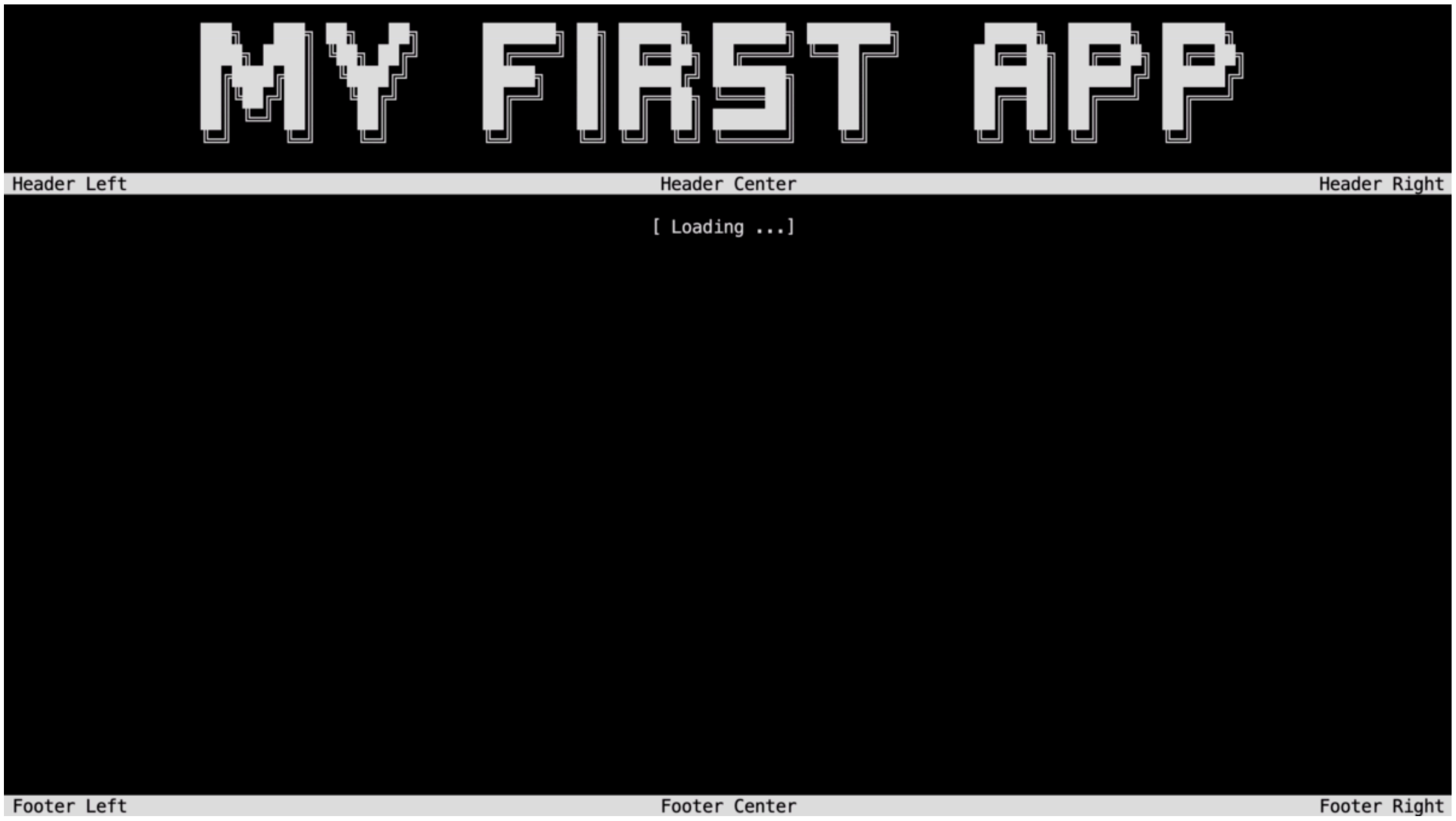


> **TIP**
>
> The method `Window.ActivateElement()` will activate the element and render it on the console. Do not forget to write `Window.Render()` before to render the other *passive* elements like `Title`, `Header` or `Footer` for example.

## Prompt

Now let's add a `Prompt` *interactive* element to your previous code and run it. Learn more

```
Prompt prompt = new Prompt("What's your name?");
Window.AddElement(prompt);

Window.ActivateElement(prompt);
```


```
MY FIRST APP
Header Left                         Header Center                         Header Right
                                    [ Loading ...]




Footer Left                         Footer Center                         Footer Right
```

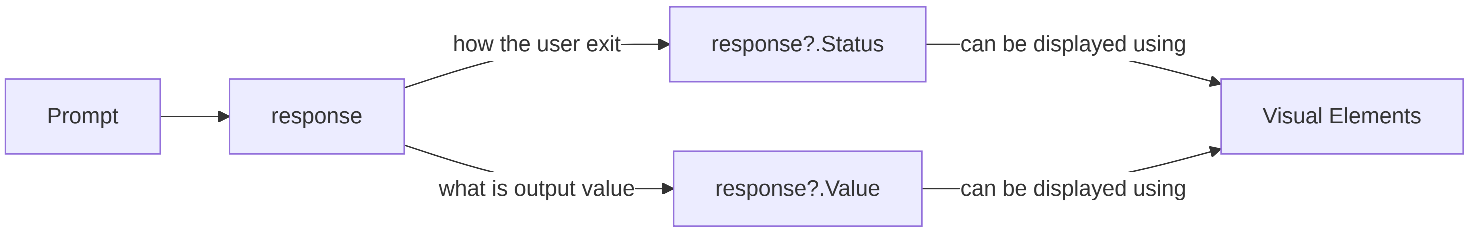
## Get response and `Dialog` element

A `Prompt` element will give you different interaction information. To get these information, let's add the following line of code after the `Window.ActivateElement(prompt)` line. [Learn more about `var`](#)

```
var response = prompt.GetResponse();
```

The previous line will retrieve a response object that has the following properties:

- `Status`: is how the interaction ended. It can be `Selected` (pressed enter), `Deleted` (pressed delete) or `Escaped` (pressed escape). It is accessible using: `response?.Status`.
- `Value`: is the user's response data. Its type depends on the `InteractiveElement` you are using. In this case, the `Prompt` element returns a `string`. It is accessible using: `response?.Value`.



**NOTE**

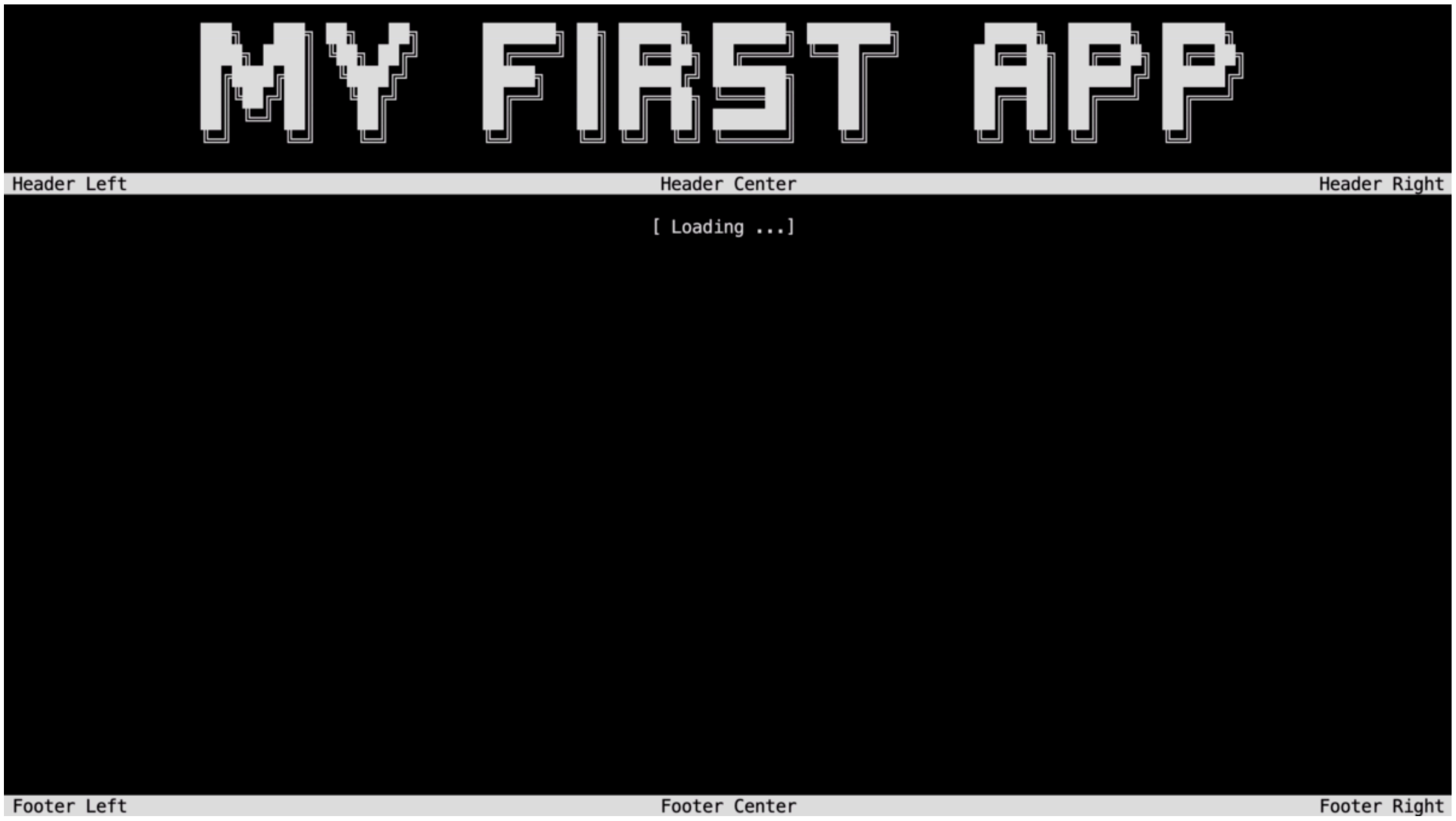
Here we use the `?.` operator to avoid a `NullReferenceException` if the response is `null`. Meaning:

- If `response` is `null`, `response?.Status` will return `null`.
- If `response` is not `null`, `response?.Status` will return `response.Status`.

If you are certain that `response` is not `null`, you can use `response!.Status` directly. This will indicate to the compiler that `response` cannot be `null` in your configuration.

Finally, let's add a `Dialog` *interactive* element to display the user's response on the console.The `Dialog` is *interactive* too and will give you a response. You may catch it if needed (as you may create an element with 0, 1 or 2 options, it will give you the selected option). Learn more

```
Dialog text = new Dialog(
    new List<string>()
    {
        "You just wrote " + response!.Value + "!",
        "And you " + response!.Status + "!"
    },
    null,
    "OK"
    );
Window.AddElement(text);
Window.ActivateElement(text);
```

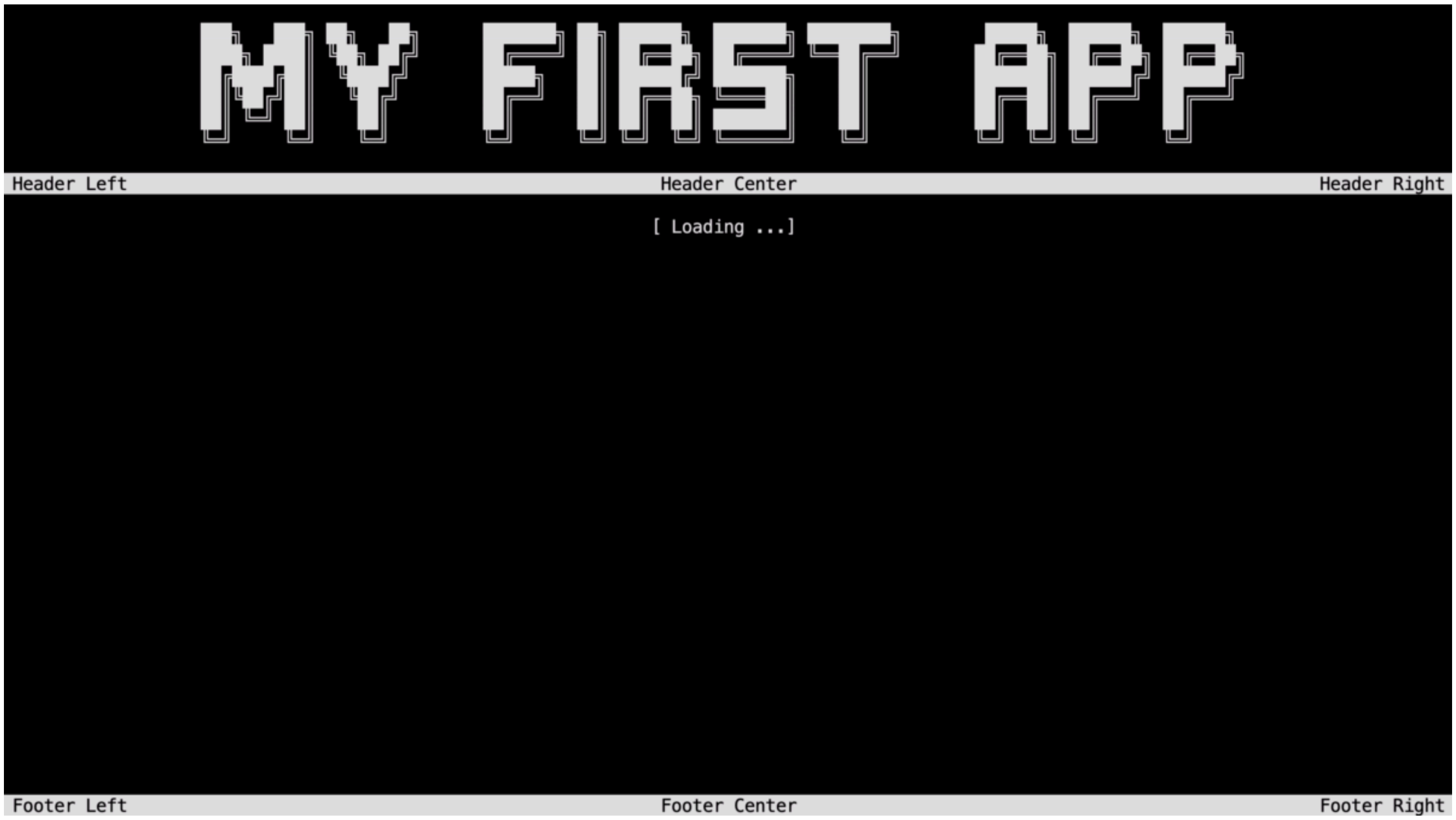


```
MY FIRST APP
Header Left                Header Center                Header Right
                           [ Loading ...]
Footer Left                Footer Center                Footer Right
```

## Exit the application

Finally, let's exit smoothly the application:

```
Window.Close();
```



## Conclusion

And that's it! You have created your first app using the `ConsoleAppVisuals` package. You can now run the app and see the result.

Here is the full code:

```
Window.Open();

Title title = new Title("My first app");
Window.AddElement(title);

Header header = new Header();
Footer footer = new Footer();
Window.AddElement(header, footer);

Window.Render();

FakeLoadingBar loadingBar = new FakeLoadingBar();
Window.AddElement(loadingBar);
Window.ActivateElement(loadingBar);

Prompt prompt = new Prompt("What's your name?");
Window.AddElement(prompt);
Window.ActivateElement(prompt);
```

```
var response = prompt.GetResponse();

Dialog text = new Dialog(
    new List<string>()
    {
        "You just wrote " + response!.Value + "!",
        "And you " + response!.Status + "!"
    },
    null,
    "OK"
    );
Window.AddElement(text);
Window.ActivateElement(text);

Window.Close();
```

---

Have a question, give a feedback or found a bug? Feel free to open an issue or start a discussion on the GitHub repository.

## Elements options

In this section, you will learn:

- How to deactivate/ remove elements
- How to use the `ElementsDashboard` inspector element
- How to use the `HeightSpacer` element
- Discover `Placement`, `TextAlignment` and `BordersType` enumerations
- How to use the full potential of the element options

## Setup

**WARNING**

We will add `using ConsoleAppVisuals.Enums;` to the using statements to use the `Placement` and `TextAlignment` enumerations.

And your cleaned `Program.cs` file should look like this:

```
using ConsoleAppVisuals;
using ConsoleAppVisuals.PassiveElements;
using ConsoleAppVisuals.InteractiveElements;
using ConsoleAppVisuals.Enums;

namespace MyApp
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Window.Open();
        }
    }
}
```

## Disabling elements

We tackled adding elements to the window. Now, let's see how to do the opposite.

To disable element rendering, you have two options:

- Deactivate the element

- Remove the element

## Deactivating

Deactivating an element can be useful for it to be used later. To do so, let's create a `Title` element and deactivate it. Nothing will be rendered on the screen.

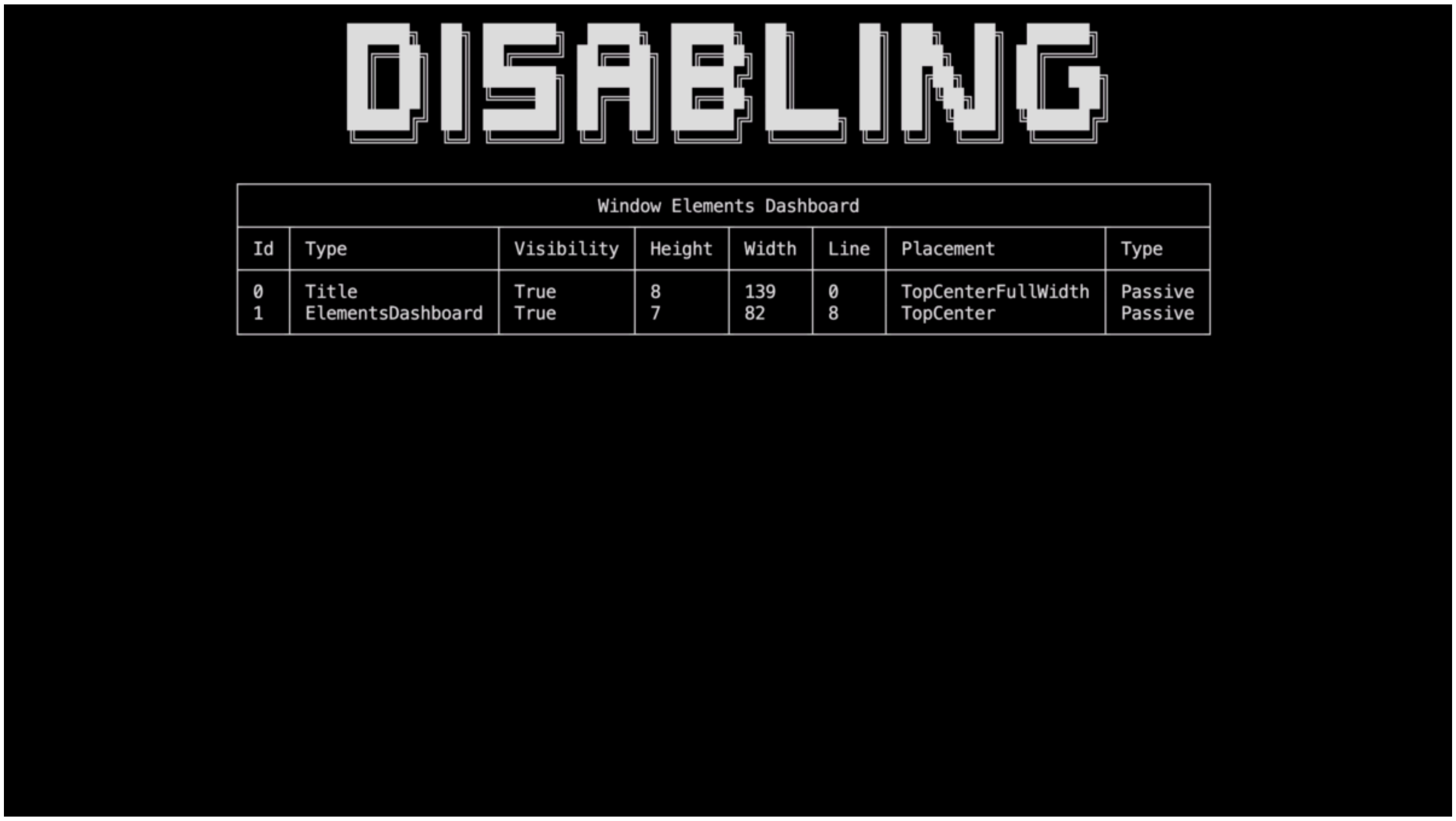
```
Window.Open();

Title title = new Title("Elements options");
Window.AddElement(title);

Window.DeactivateElement(title);

Window.Render();

Window.Close();
```

Let's see how to perceive the effect of deactivating an element. Update your code to add a `ElementsDashboard` *passive* element and deactivate the title. The dashboard will be rendered, but not the title:

**NOTE**

The method `Window.Freeze()` is used to stop the execution by waiting the user to press a key (Enter by default) to see the window content without exiting the application when the window only contains *passive* elements.

```
Window.Open();

Title title = new Title("Elements options");
Window.AddElement(title);

ElementsDashboard dashboard = new ElementsDashboard();
Window.AddElement(dashboard);

Window.Render();
Window.Freeze();

Window.DeactivateElement(title);

Window.Render();
Window.Freeze();

Window.Close();
```



```
┌─────────────────────────────────────────────────────────────────────────────────────────┐
│                               Window Elements Dashboard                                 │
├────┬───────────────────┬────────────┬────────┬───────┬──────┬───────────────────┬─────────┤
│ Id │ Type              │ Visibility │ Height │ Width │ Line │ Placement         │ Type    │
├────┼───────────────────┼────────────┼────────┼───────┼──────┼───────────────────┼─────────┤
│ 0  │ Title             │ True       │ 8      │ 139   │ 0    │ TopCenterFullWidth│ Passive │
│ 1  │ ElementsDashboard │ True       │ 7      │ 82    │ 8    │ TopCenter         │ Passive │
└────┴───────────────────┴────────────┴────────┴───────┴──────┴───────────────────┴─────────┘
```

As you noticed, the title is not rendered on the screen because its `Visibility` property has been set to false.

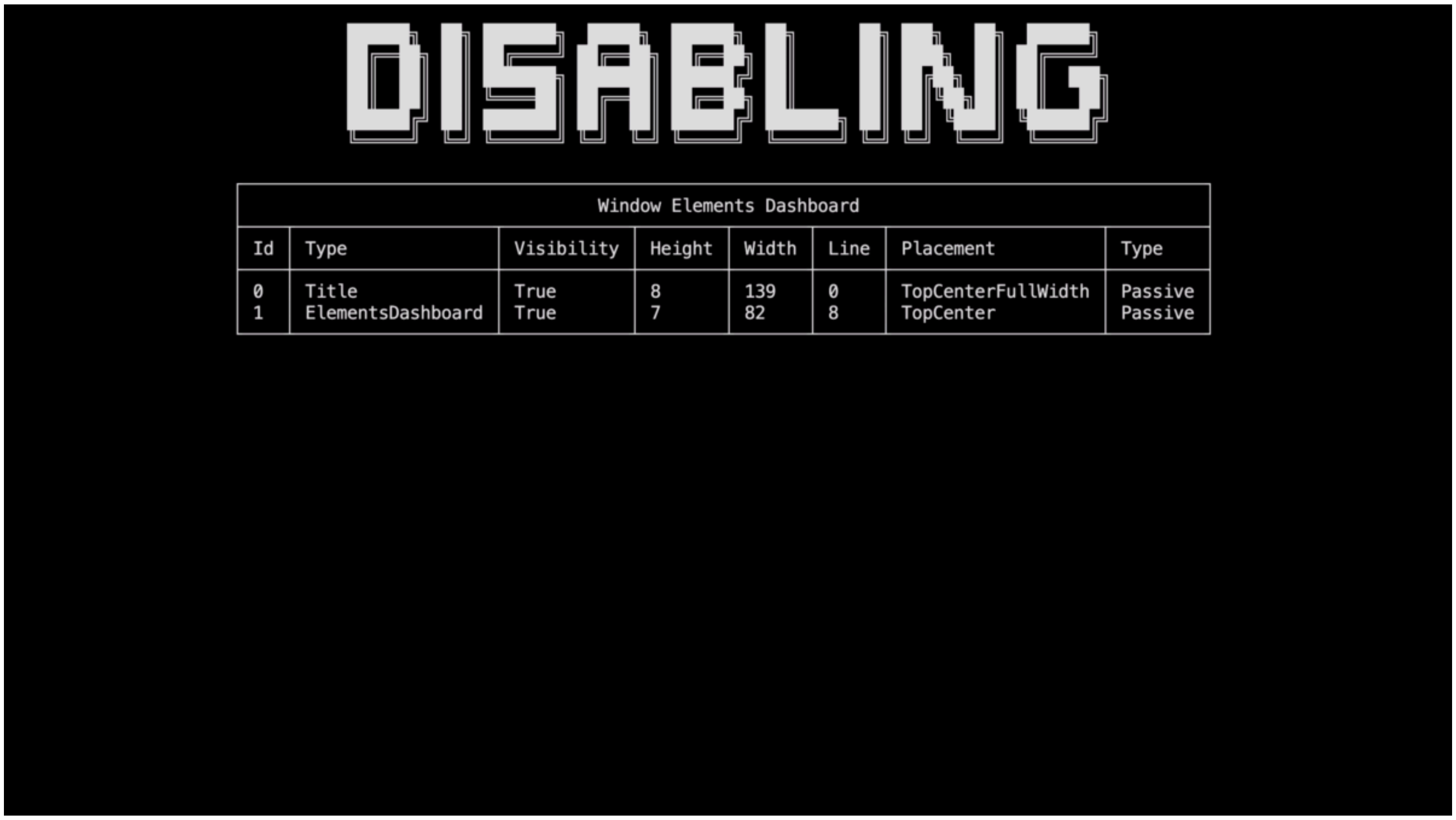
## Removing

Removing an element is useful when you don't want to use it anymore. To do so, let's create a `Title` element and remove it. Nothing will be rendered on the screen.

```
Window.Open();

Title title = new Title("Disabling");
Window.AddElement(title);

Window.RemoveElement(title);

Window.Render();

Window.Close();
```

Let's see how to perceive the effect of removing an element. Update your code to the following:

```
Window.Open();

Title title = new Title("Disabling");
Window.AddElement(title);

ElementsDashboard dashboard = new ElementsDashboard();
Window.AddElement(dashboard);

Window.Render();
Window.Freeze();

Window.RemoveElement(title);

Window.Render();
Window.Freeze();

Window.Close();
```



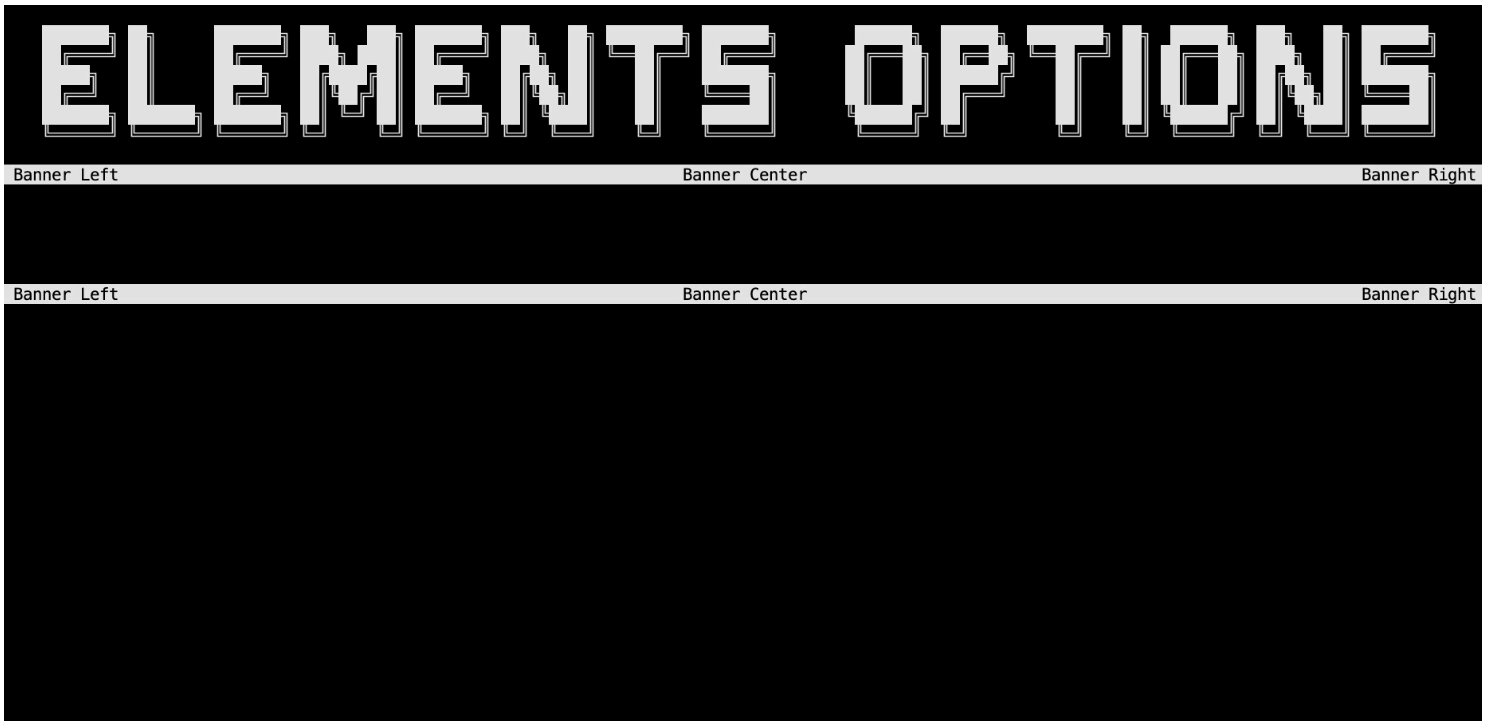
```
┌──────────────────────────────────────────────────────────────────────────────────────┐
│                               Window Elements Dashboard                              │
├────┬───────────────────┬────────────┬────────┬───────┬──────┬───────────────────┬─────────┤
│ Id │ Type              │ Visibility │ Height │ Width │ Line │ Placement         │ Type    │
├────┼───────────────────┼────────────┼────────┼───────┼──────┼───────────────────┼─────────┤
│ 0  │ Title             │ True       │ 8      │ 139   │ 0    │ TopCenterFullWidth│ Passive │
│ 1  │ ElementsDashboard │ True       │ 7      │ 82    │ 8    │ TopCenter         │ Passive │
└────┴───────────────────┴────────────┴────────┴───────┴──────┴───────────────────┴─────────┘
```

## Setting a height spacer between elements

Sometimes, you may want to add a space between elements. To do so, you can use the `HeightSpacer` element. It is a *passive* element that will only render a space of a specific height between elements.

Here is a concrete example between two banners:

```
Window.Open();

Title title = new Title("Elements options");
Window.AddElement(title);

Banner banner1 = new Banner();
HeightSpacer spacer = new HeightSpacer(5);
Banner banner2 = new Banner();

Window.AddElement(banner1, spacer, banner2);

Window.Render();
Window.Freeze();

Window.Close();
```



```
ELEMENTS OPTIONS
Banner Left                                         Banner Center                                     Banner Right





Banner Left                                         Banner Center                                     Banner Right
```

**NOTE**

You may update afterward the `Placement` and `Height` of the `HeightSpacer` element using the `UpdatePlacement()` and `UpdateHeight()` methods.

## Access and update elements parameters

In all the tutorials and the [example project](example project) the elements definitions are simplified and do not declare all the arguments available. To see all the arguments available for each element, you can consult the [references documentation](references documentation).

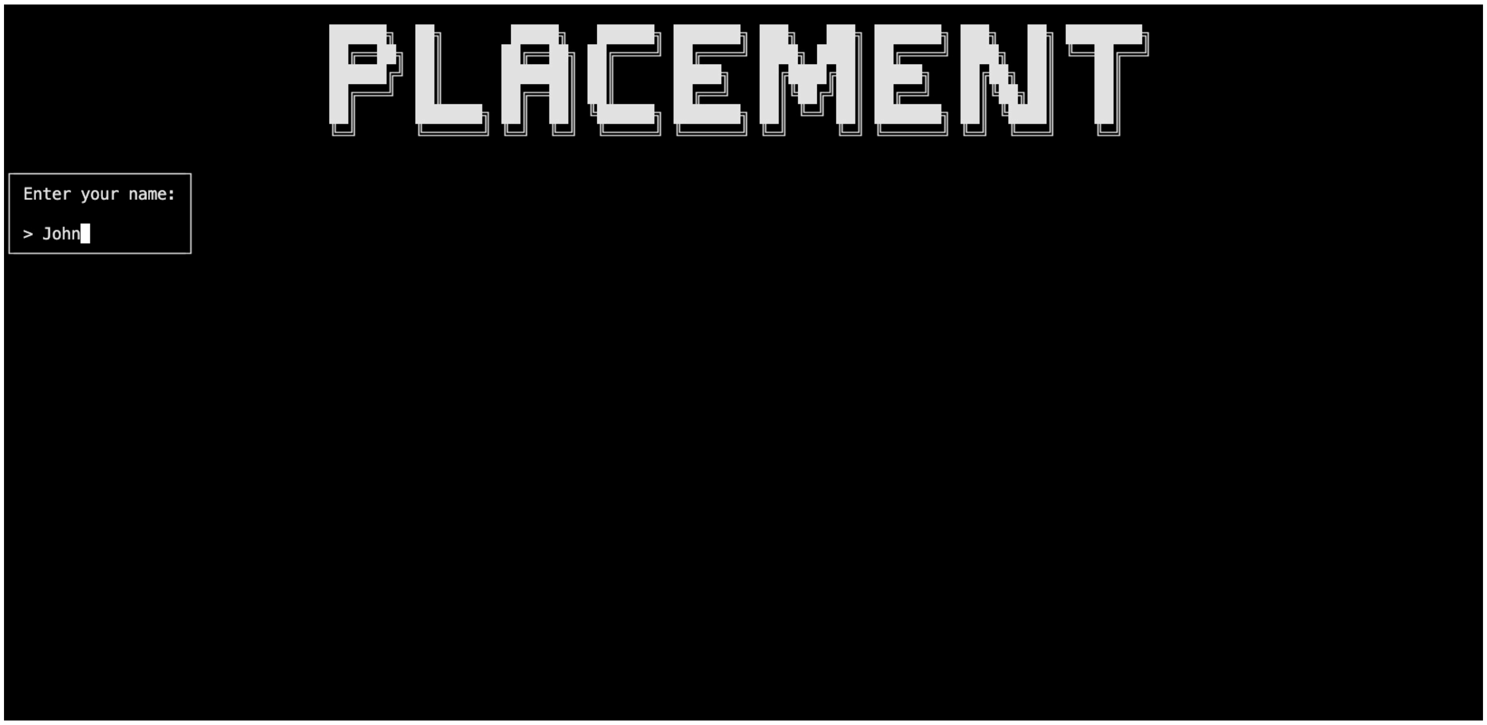
Most of them are specific with generic type (`string`, `int`, `bool`, `List`, ...) and are used to customize the element. But some of them are common to all elements and are used to place the element on the window. These are the `Placement` and `TextAlignment` enumerations.

### `Placement`

The `Placement` enumeration is used to place elements at a convenient location on the window. It is used by every element from the library that can be placed on the window. According to the placement, the element position and line will be calculated and rendered.

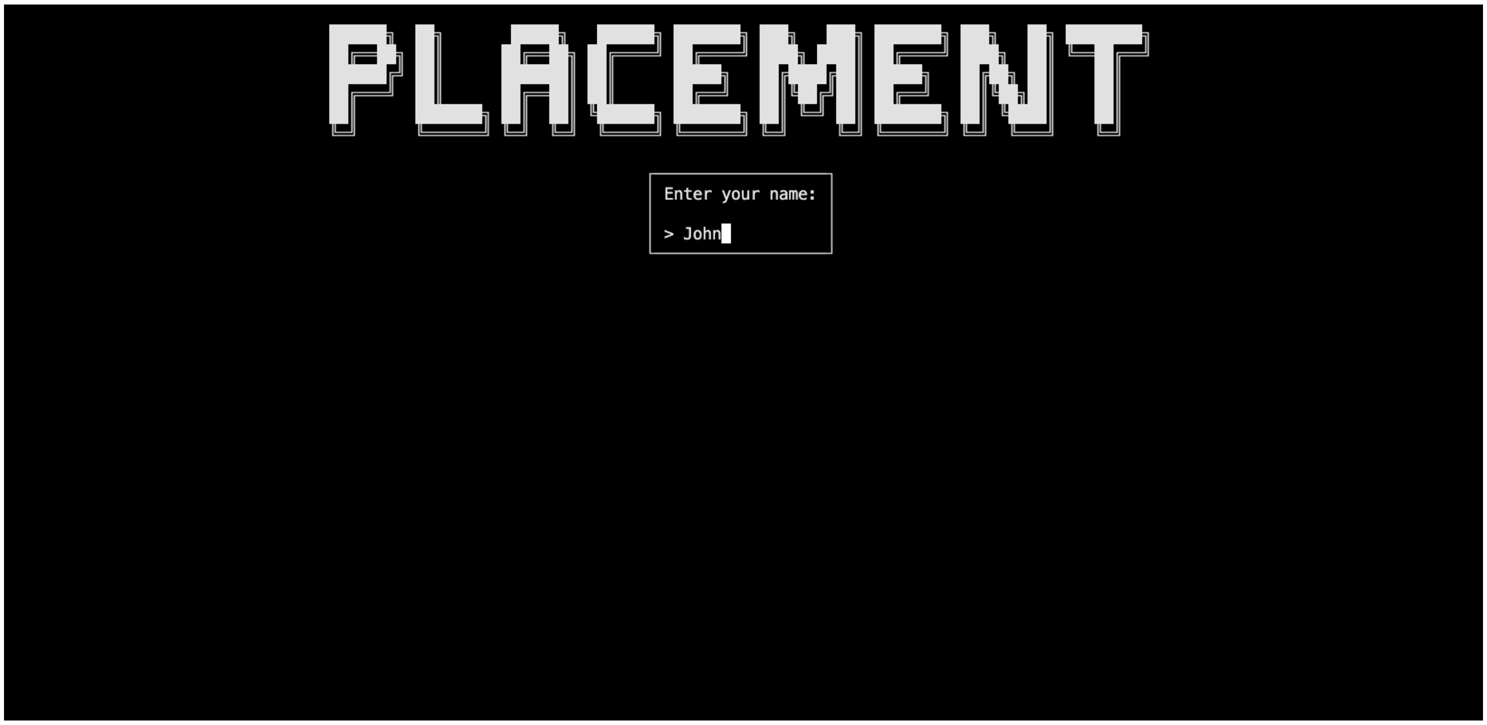
The available values are:

- `TopLeft`: x(line) = 0, y(char) = 0



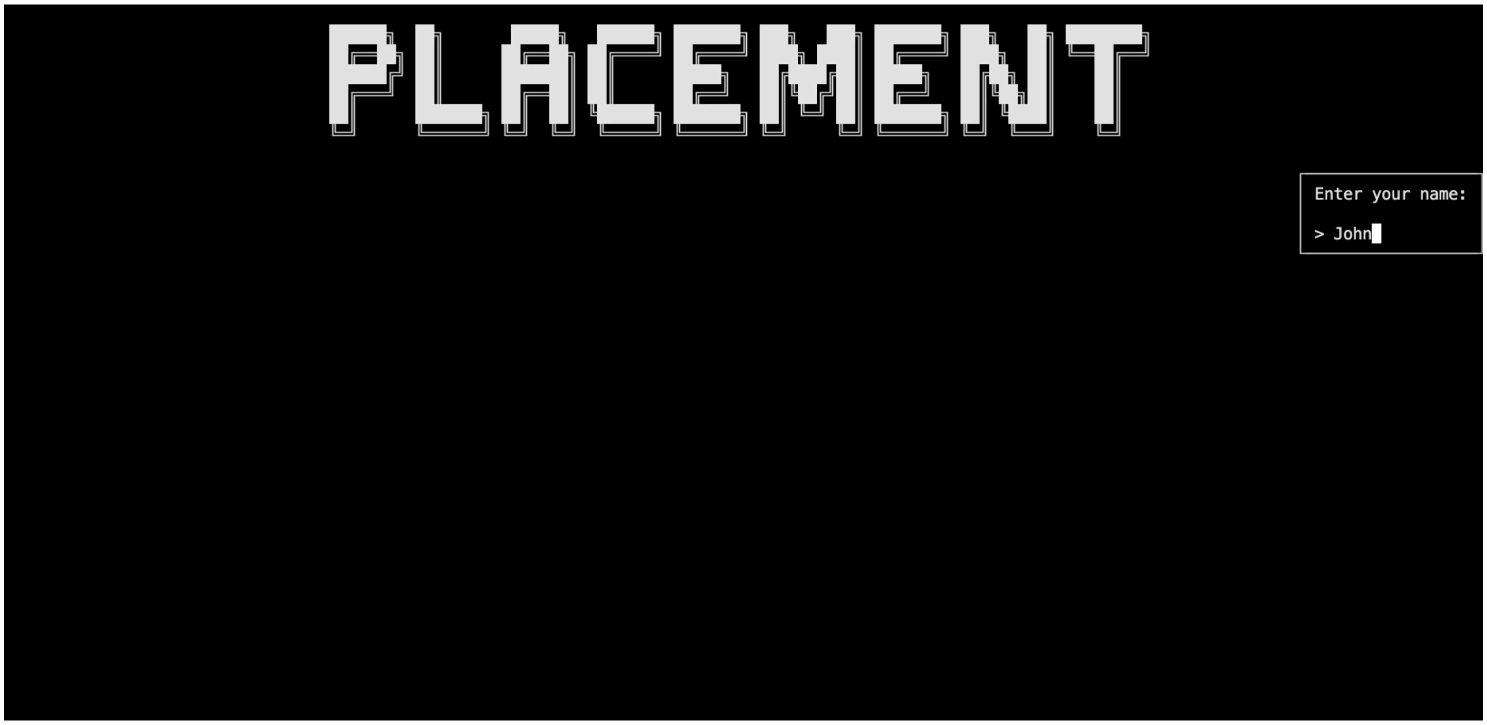
```
Enter your name:

> John
```

- `TopCenter`: (Default) x(line) = 0, y(char) = windowWidth / 2



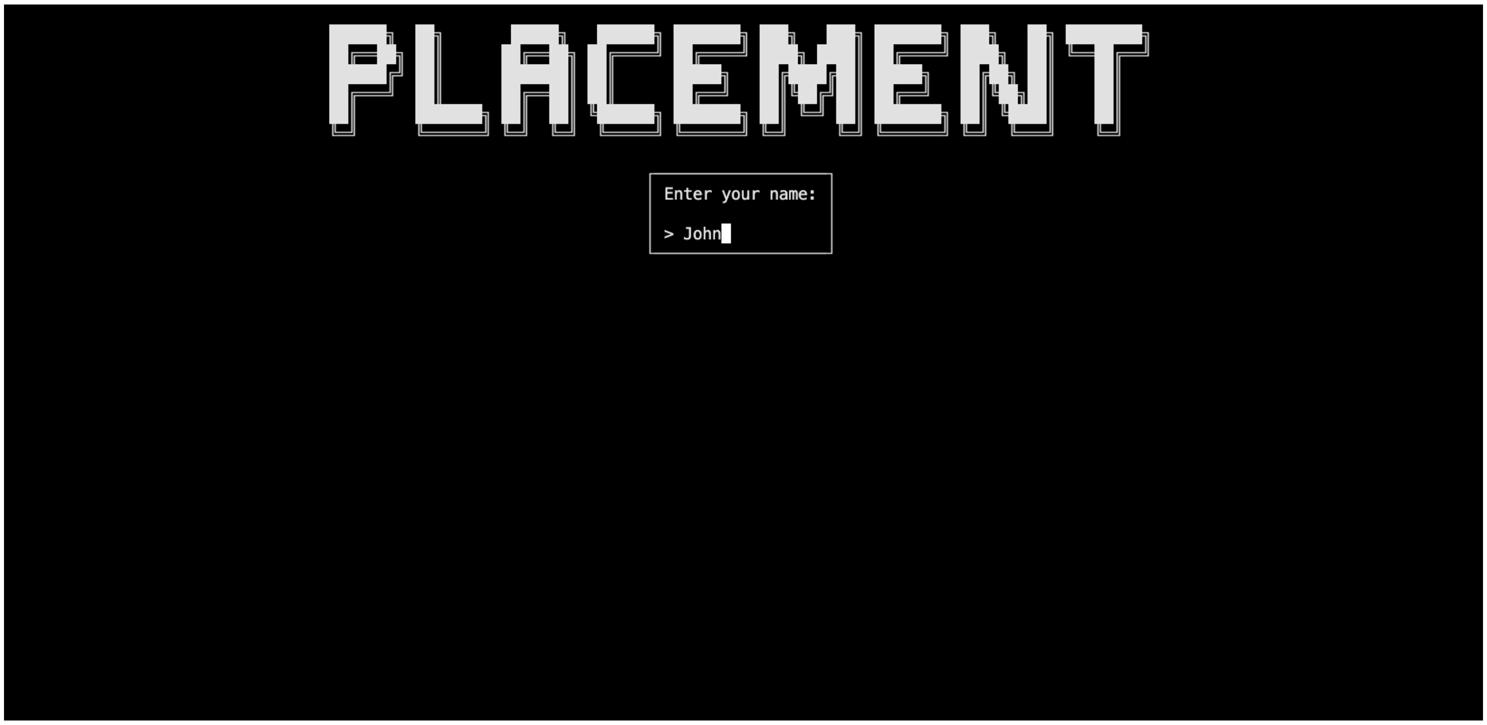
```
Enter your name:

> John
```

- `TopRight`: x(line) = 0, y(char) = windowWidth



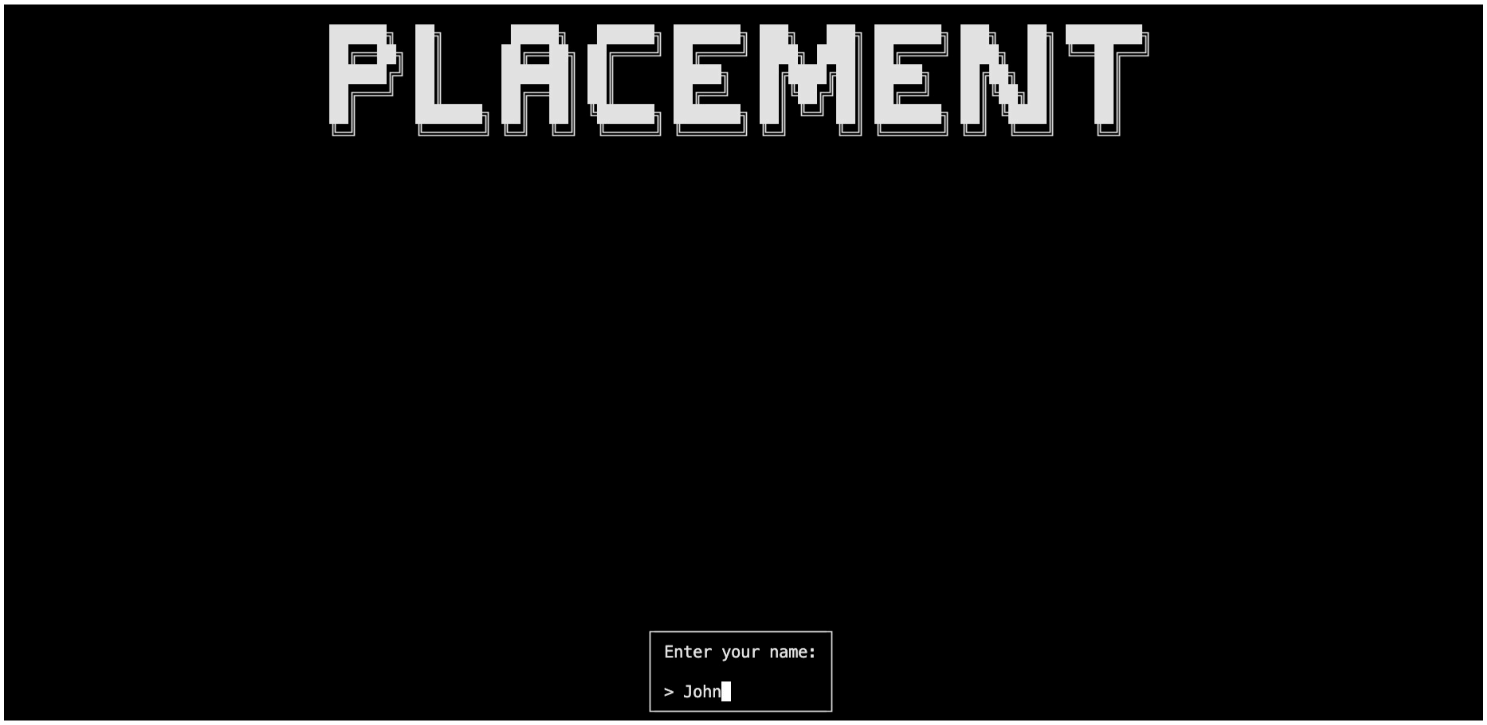
```
Enter your name:

> John
```

- `TopCenterFullWidth`: x(line) = 0, y(char) = 0 (In fact, it is the same as `TopLeft` but we know that the element will be rendered with the full width of the window, following top elements will be placed below it)



```
Enter your name:

> John
```

- `BottomCenterFullWidth`: x(line) = windowHeight, y(char) = 0 (In preview for now as not fully implemented)



```
Enter your name:

> John
```

**NOTE**

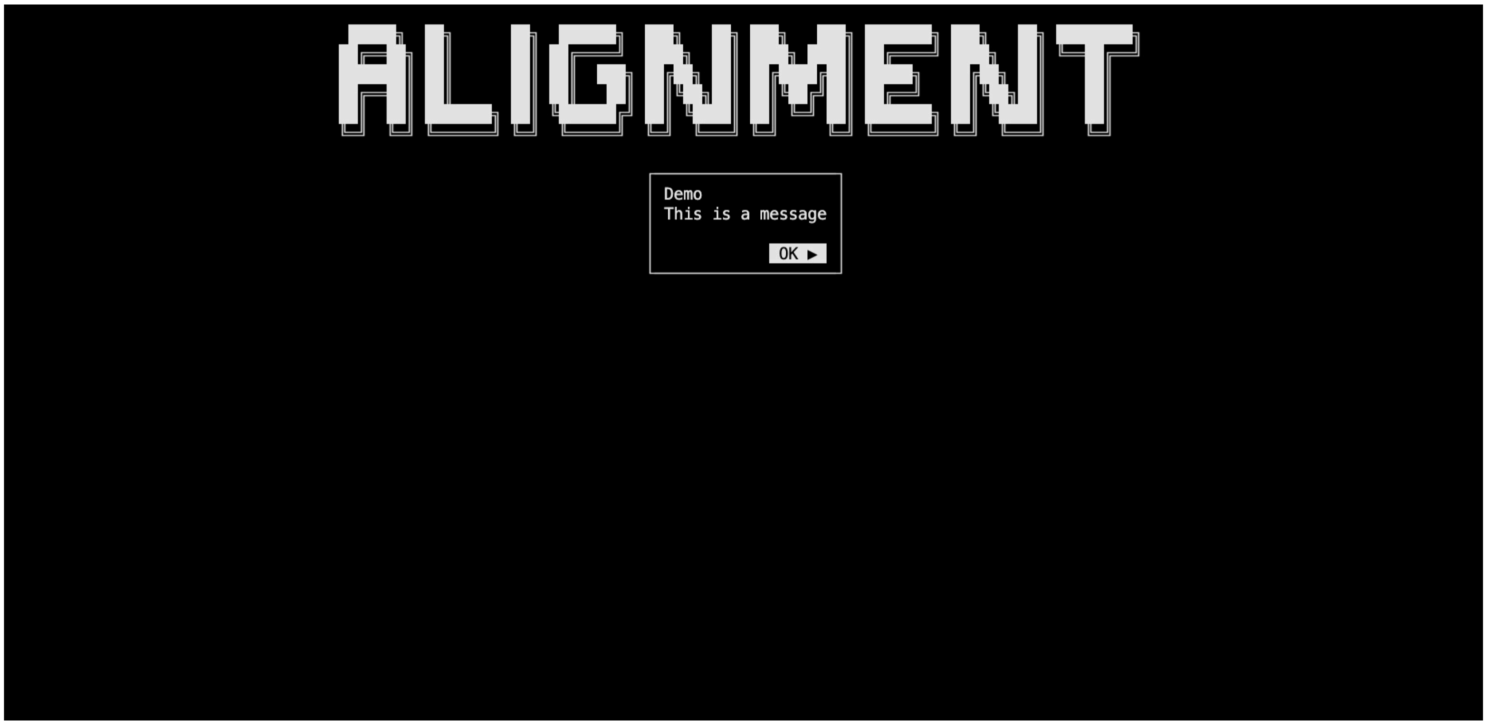
To choose the placement of an element, you can either set it from the constructor or use the `UpdatePlacement()` method after creating the element.

```
Prompt prompt = new Prompt("Enter your name:", "John", Placement.TopCenter);
// or
prompt.UpdatePlacement(Placement.TopCenter);
```

## TextAlignment

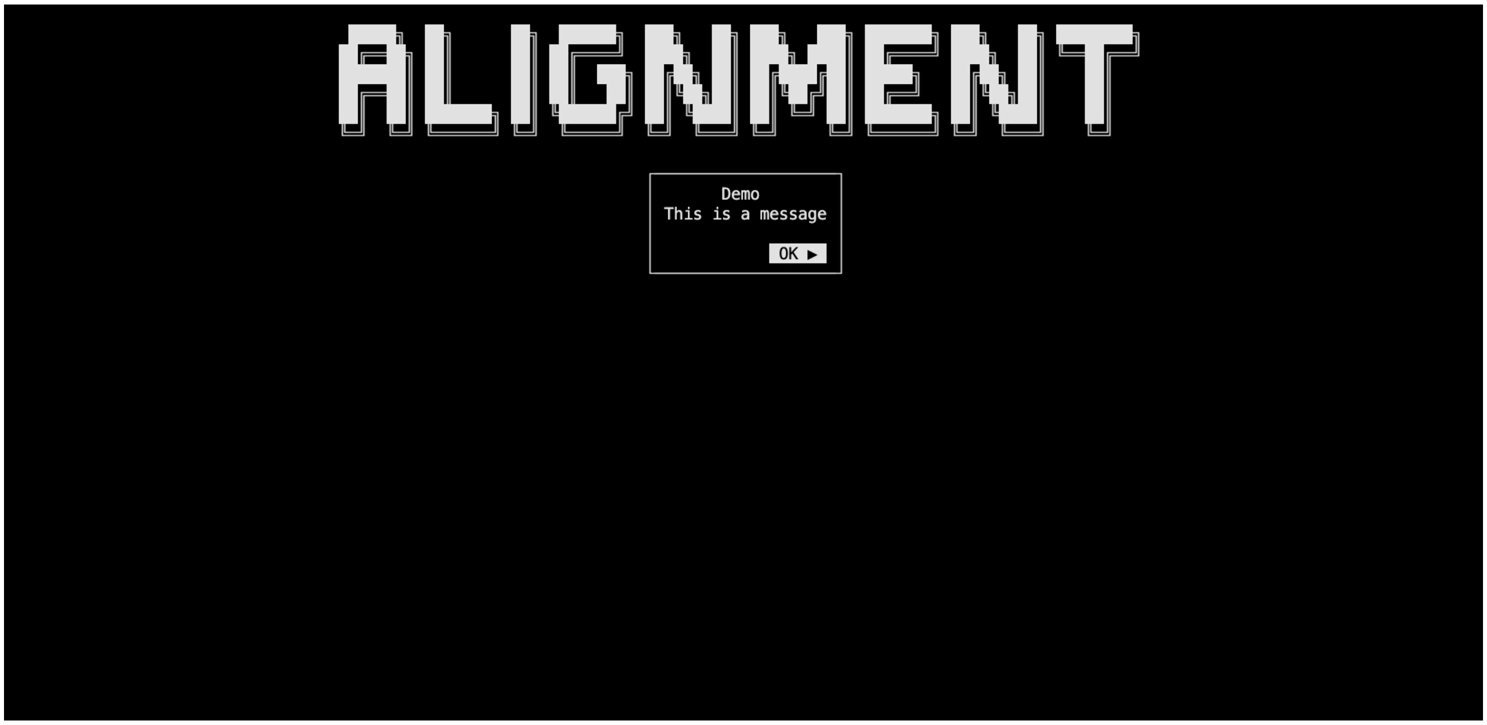
The `TextAlignment` enumeration is used to align the text in a string. It is used by some elements from the library. Here are the available values:

- `Left`: Align the text to the left



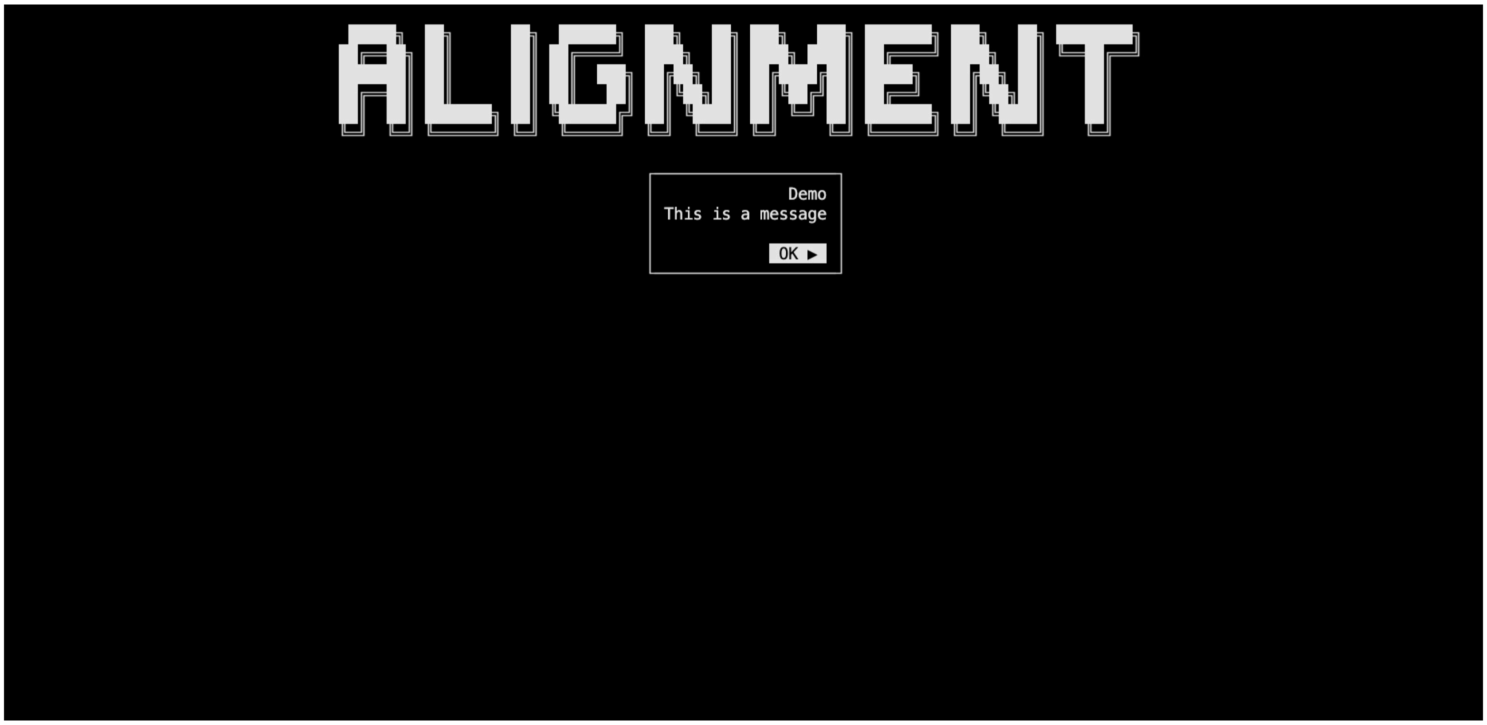
```
Demo
This is a message
               OK ▶
```

- `Center`: (Default) Align the text to the center



```
      Demo
This is a message

          OK ▶
```

- `Right`: Align the text to the right



```
             Demo
This is a message

          OK ▶
```

**NOTE**

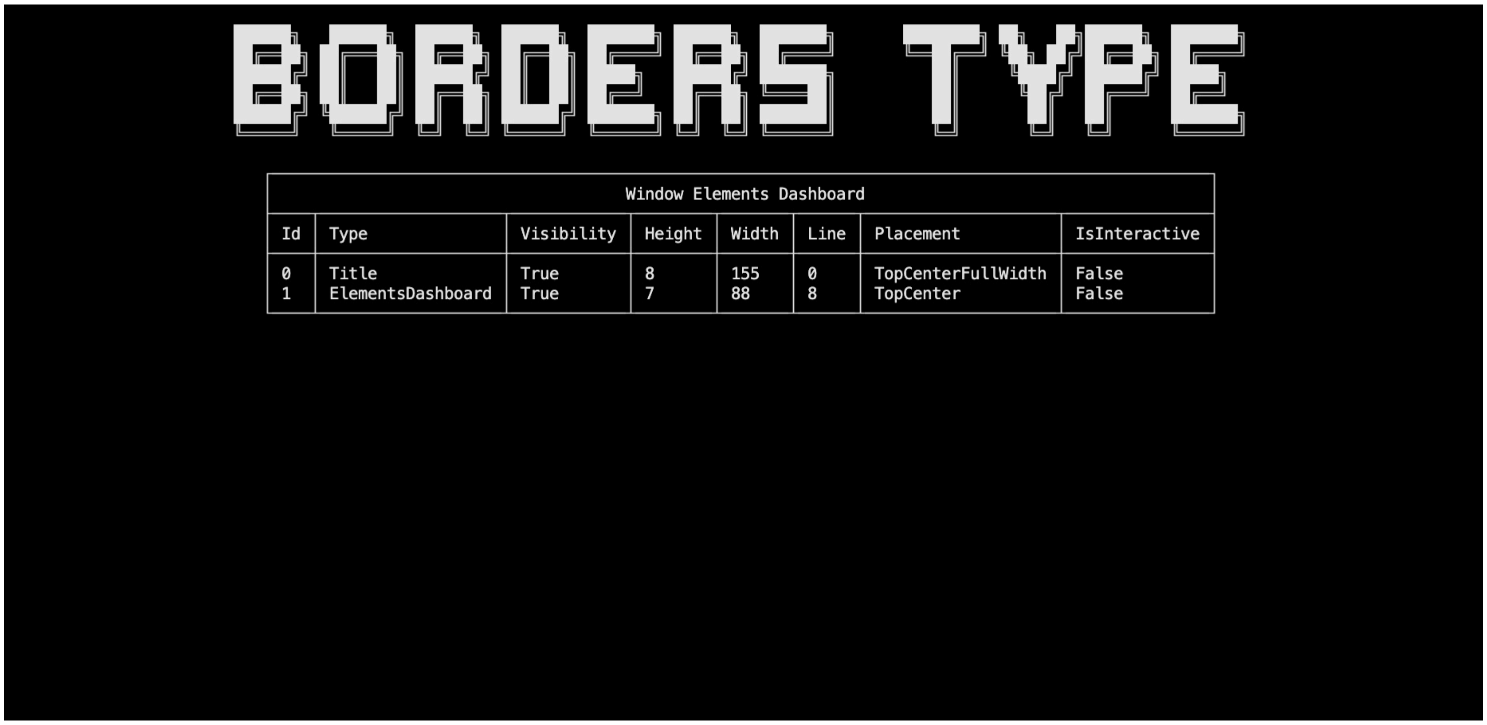
To choose the text alignment of an element, you can either set it from the constructor or use the `UpdateTextAlignment()` method after creating the element (some elements may not have this method if the text alignment is not used in it so refer to the references documentation to get that specific information).

```
Dialog dialog = new Dialog(new List<string>(){"Demo", "This is a message"},
null, "OK ►", TextAlignment.Center);
// or
dialog.UpdateTextAlignment(TextAlignment.Center);
```

## BordersType

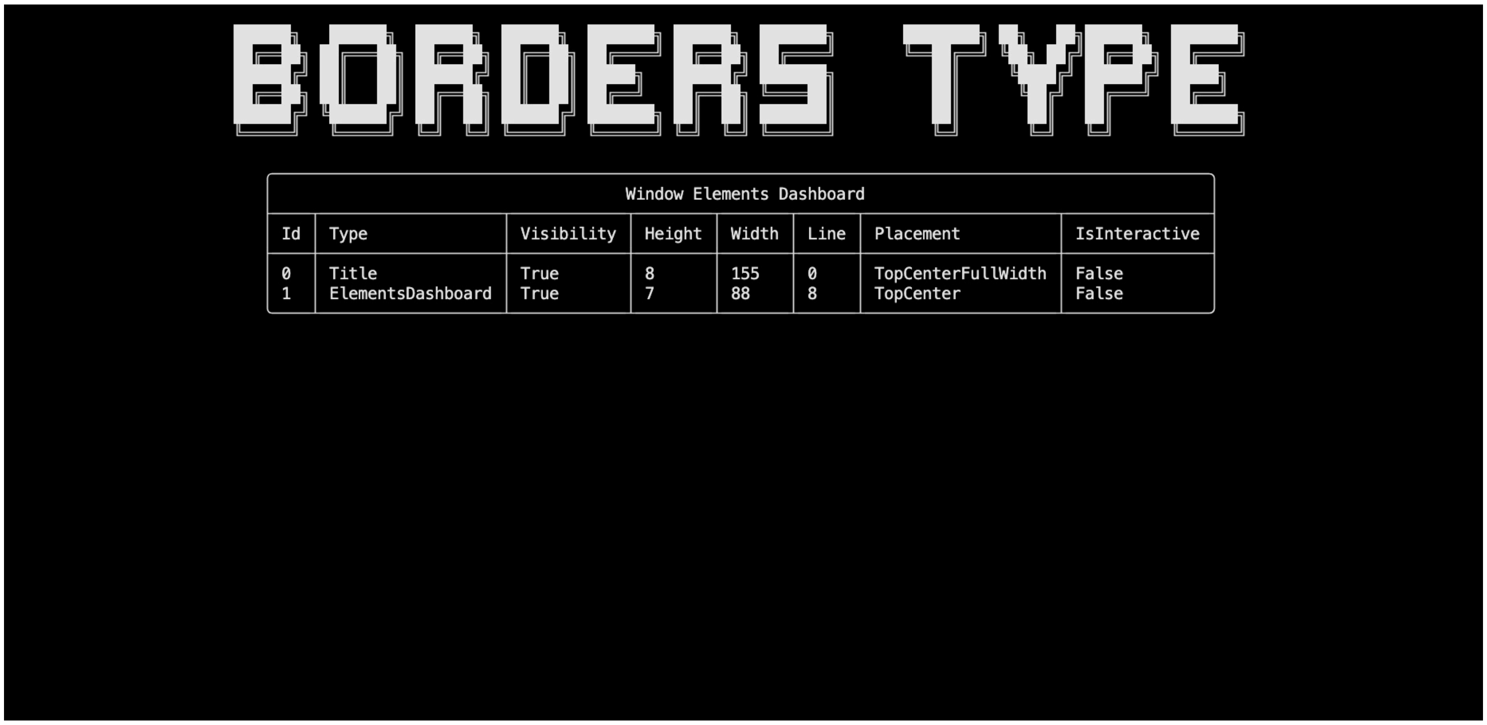
The `BordersType` enumeration is used to set the borders of an element. It is used by the table and embed elements from the library. Here are the available values:

- `SingleStraight`: (Default) Single lines with straight corners



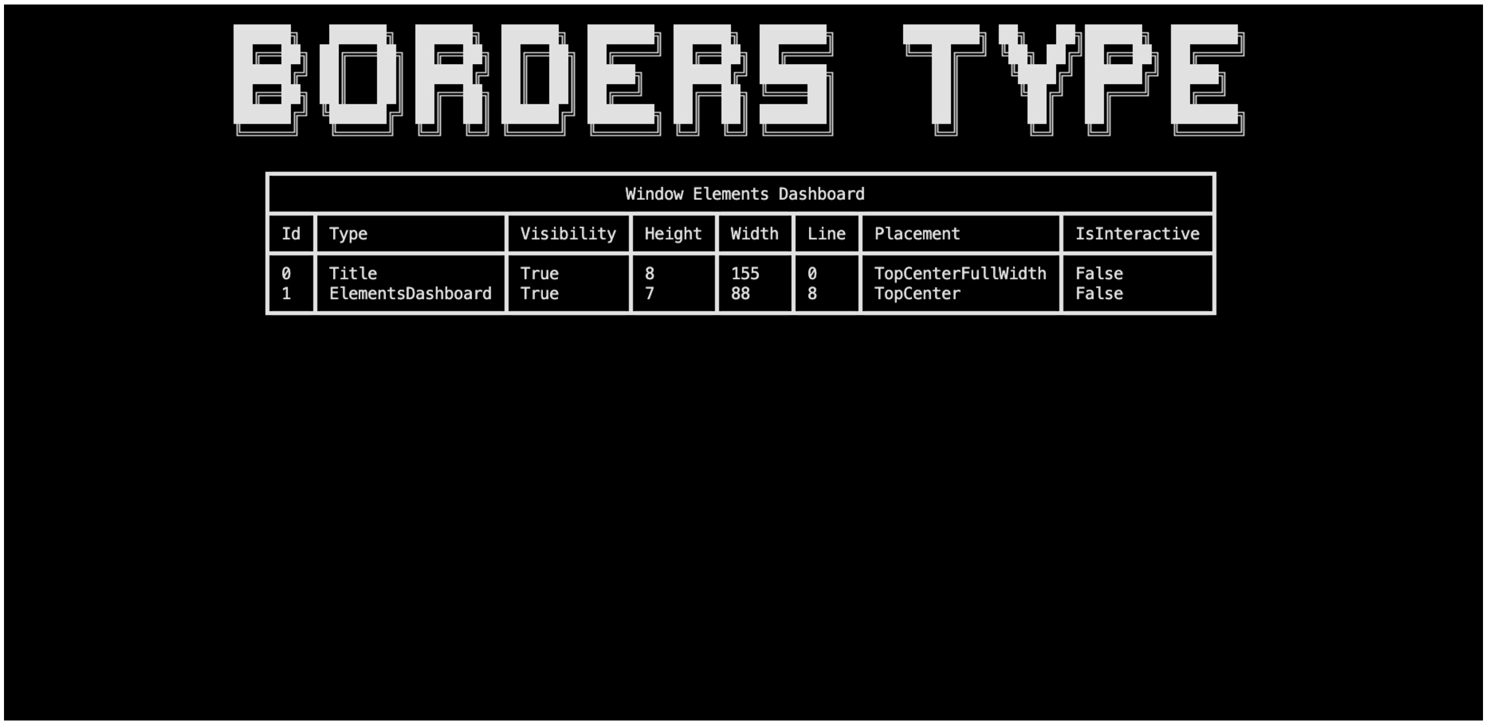
```
BORDERS TYPE

┌──────────────────────────────────────────────────────────────────────────────────────────────┐
│                                 Window Elements Dashboard                                    │
├────┬──────────────────┬────────────┬────────┬───────┬──────┬───────────────────┬─────────────┤
│ Id │ Type             │ Visibility │ Height │ Width │ Line │ Placement         │ IsInteractive│
├────┼──────────────────┼────────────┼────────┼───────┼──────┼───────────────────┼─────────────┤
│ 0  │ Title            │ True       │ 8      │ 155   │ 0    │ TopCenterFullWidth│ False       │
│ 1  │ ElementsDashboard│ True       │ 7      │ 88    │ 8    │ TopCenter         │ False       │
└────┴──────────────────┴────────────┴────────┴───────┴──────┴───────────────────┴─────────────┘
```

- `SingleRound`: Single lines with round corners



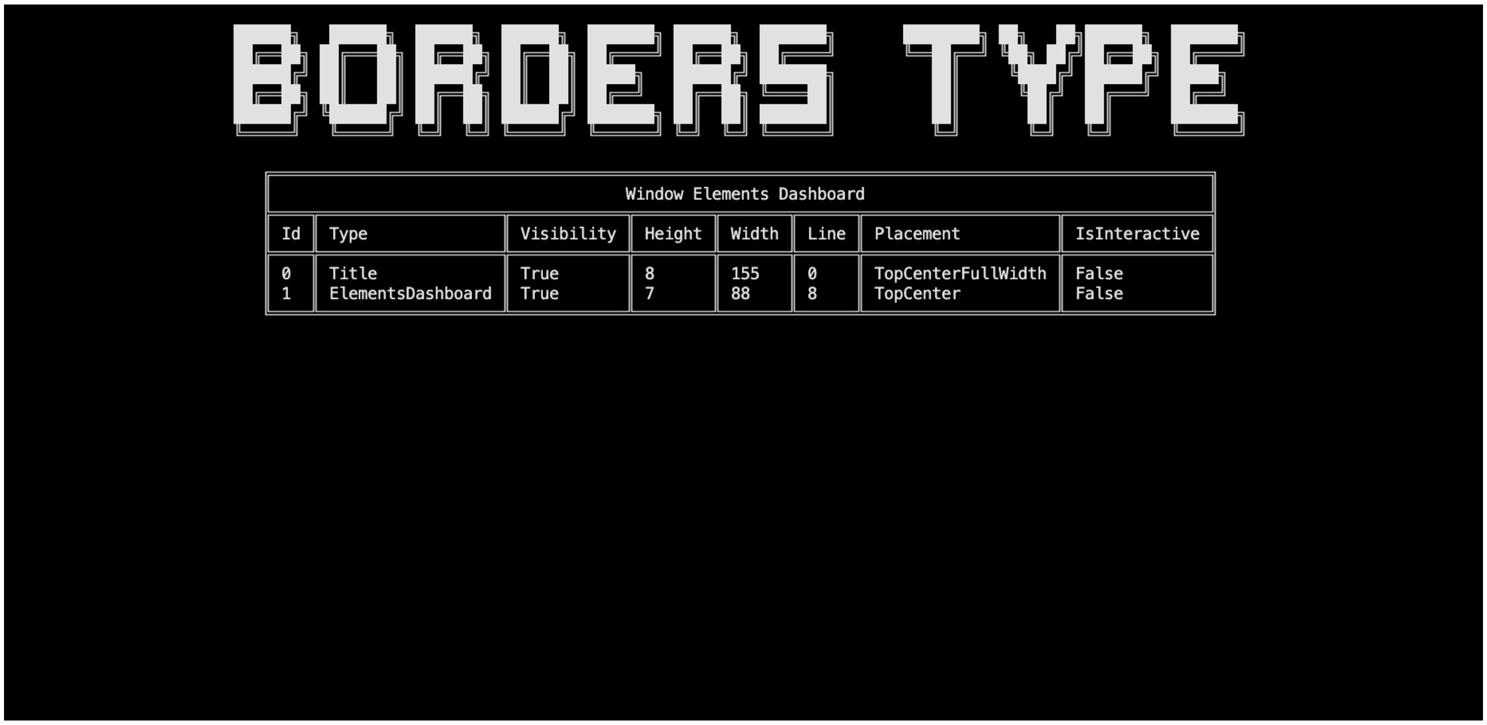
| Window Elements Dashboard | | | | | | | |
|---|---|---|---|---|---|---|---|
| Id | Type | Visibility | Height | Width | Line | Placement | IsInteractive |
| 0 | Title | True | 8 | 155 | 0 | TopCenterFullWidth | False |
| 1 | ElementsDashboard | True | 7 | 88 | 8 | TopCenter | False |

- `SingleBold`: Single bold lines with straight corners



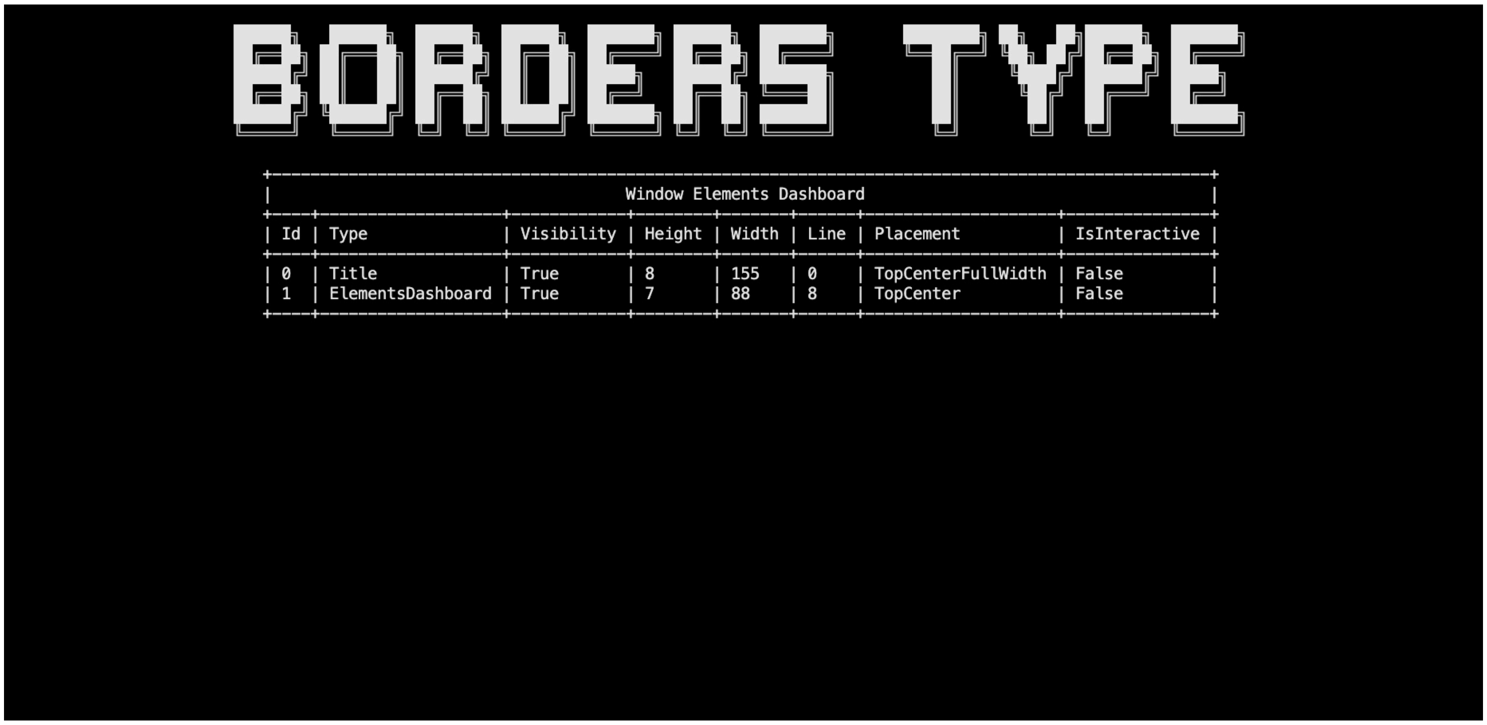
| Window Elements Dashboard | | | | | | | |
|---|---|---|---|---|---|---|---|
| Id | Type | Visibility | Height | Width | Line | Placement | IsInteractive |
| 0 | Title | True | 8 | 155 | 0 | TopCenterFullWidth | False |
| 1 | ElementsDashboard | True | 7 | 88 | 8 | TopCenter | False |

- `DoubleStraight`: Double lines with straight corners



```
BORDERS TYPE

╔═════════════════════════════════════════════════════════════════════════════════════════════════╗
║                                    Window Elements Dashboard                                    ║
╠════╦═══════════════════╦════════════╦════════╦═══════╦══════╦═══════════════════╦═══════════════╣
║ Id ║ Type              ║ Visibility ║ Height ║ Width ║ Line ║ Placement         ║ IsInteractive ║
╠════╬═══════════════════╬════════════╬════════╬═══════╬══════╬═══════════════════╬═══════════════╣
║ 0  ║ Title             ║ True       ║ 8      ║ 155   ║ 0    ║ TopCenterFullWidth║ False         ║
║ 1  ║ ElementsDashboard ║ True       ║ 7      ║ 88    ║ 8    ║ TopCenter         ║ False         ║
╚════╩═══════════════════╩════════════╩════════╩═══════╩══════╩═══════════════════╩═══════════════╝
```

- `ASCII`: ASCII basic characters for the borders (`+`, `-`, `|` only)



```
BORDERS TYPE

+------------------------------------------------------------------------------------------------+
|                                   Window Elements Dashboard                                    |
+----+-------------------+------------+--------+-------+------+--------------------+---------------+
| Id | Type              | Visibility | Height | Width | Line | Placement          | IsInteractive |
+----+-------------------+------------+--------+-------+------+--------------------+---------------+
| 0  | Title             | True       | 8      | 155   | 0    | TopCenterFullWidth | False         |
| 1  | ElementsDashboard | True       | 7      | 88    | 8    | TopCenter          | False         |
+----+-------------------+------------+--------+-------+------+--------------------+---------------+
```

**WARNING**

The following types may not work on Visual Studio or Windows Command Prompt:

- `SingleRound`
- `SingleBold`

**NOTE**

To choose the border type of an element, you can either set it from the constructor or use the `UpdateBordersType()` method after creating the element (some elements may not have this method if the border type is not used in it so refer to the references documentation to get that specific information).

```
ElementsDashboard dashboard = new ElementsDashboard(Placement.TopCenter,
BordersType.SingleStraight);
// or
dashboard.UpdateBordersType(BordersType.SingleStraight);
```

## Conclusion

In this section, you learned how to deactivate and remove elements from the window. You also discovered the `Placement` and `TextAlignment` enumerations and how to use the full potential of the element options by knowing all the arguments available. You may now be able to use more complex elements and place them at your desired location.

---

Have a question, give a feedback or found a bug? Feel free to open an issue or start a discussion on the GitHub repository.

# Data visualization

In this section, you will:

- Discover data visualization with `TableView`, `TableSelector` and `Matrix` elements

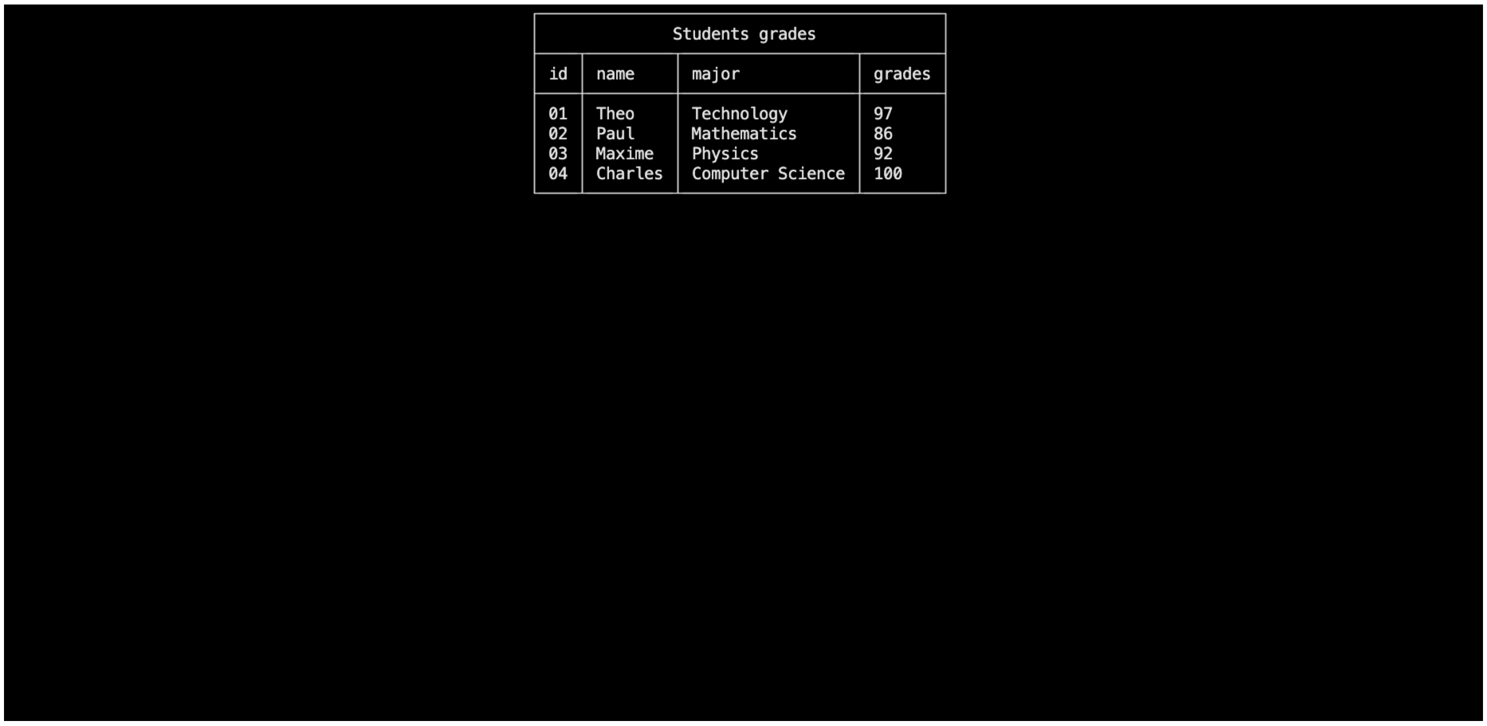
> **TIP**
>
> Each subsection is independent. I recommend you to overwrite the `Program.cs` file with the code of each section to avoid any confusion.

## The `TableView` element

The `TableView` element is used to display data in a table format. It is useful when you want to display a list of items with multiple columns. Learn more

Let's create a `TableView` element and add it to the window.

```
Window.Open();

List<string> studentsHeaders = new List<string>() { "id", "name", "major",
"grades" };

List<string> student1 = new List<string>() { "01", "Theo", "Technology", "97" };
List<string> student2 = new List<string>() { "02", "Paul", "Mathematics", "86" };
List<string> student3 = new List<string>() { "03", "Maxime", "Physics", "92" };
List<string> student4 = new List<string>() { "04", "Charles", "Computer Science",
"100" };

List<List<string>> studentsData =
    new List<List<string>>()
    {
        student1,
        student2,
        student3,
        student4
    };

TableView students =
    new TableView(
        "Students grades",
        studentsHeaders,
        studentsData
    );

Window.AddElement(students);
Window.Render(students);
Window.Freeze();

Window.Close();
```



```
┌─────────────────────────────────────────┐
│             Students grades             │
├────┬─────────┬──────────────────┬───────┤
│ id │ name    │ major            │ grades│
├────┼─────────┼──────────────────┼───────┤
│ 01 │ Theo    │ Technology       │ 97    │
│ 02 │ Paul    │ Mathematics      │ 86    │
│ 03 │ Maxime  │ Physics          │ 92    │
│ 04 │ Charles │ Computer Science │ 100   │
└────┴─────────┴──────────────────┴───────┘
```

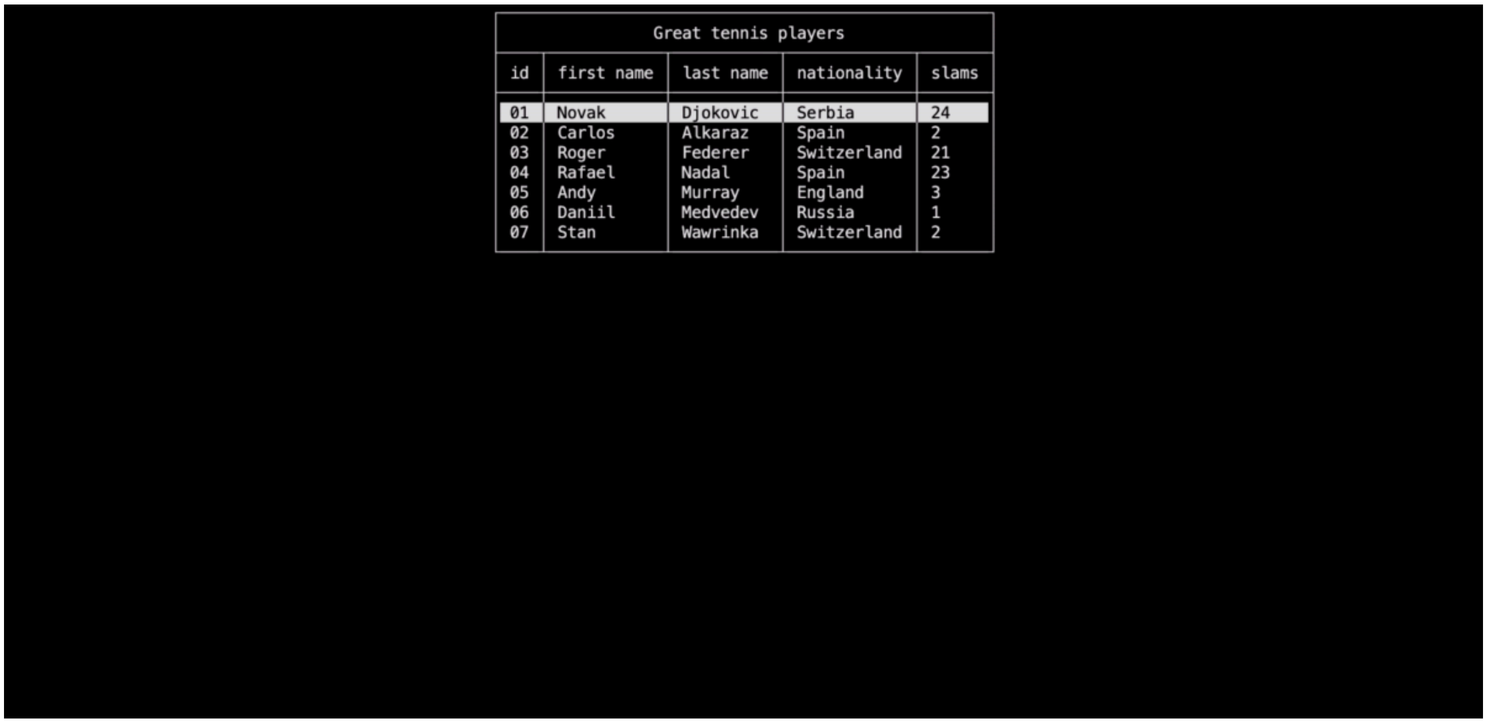
## The `TableSelector` element

The `TableSelector` element is used to display data in a table format and allow the user to select a row. It is useful when you want to be able to interact with a table. You may use the `Up` and `Down` arrows or the `Z` and `S` keys to move int the selector. [Learn more](#)

Here is an example of how to use it:

```
Window.Open();

List<string> playersHeaders = new List<string>() { "id", "first name", "last name",
"nationality", "slams" };

List<string> player1 = new List<string>() { "01", "Novak", "Djokovic", "Serbia",
"24" };
List<string> player2 = new List<string>() { "02", "Carlos", "Alkaraz", "Spain",
"2" };
List<string> player3 = new List<string>() { "03", "Roger", "Federer", "Switzerland",
"21" };
```

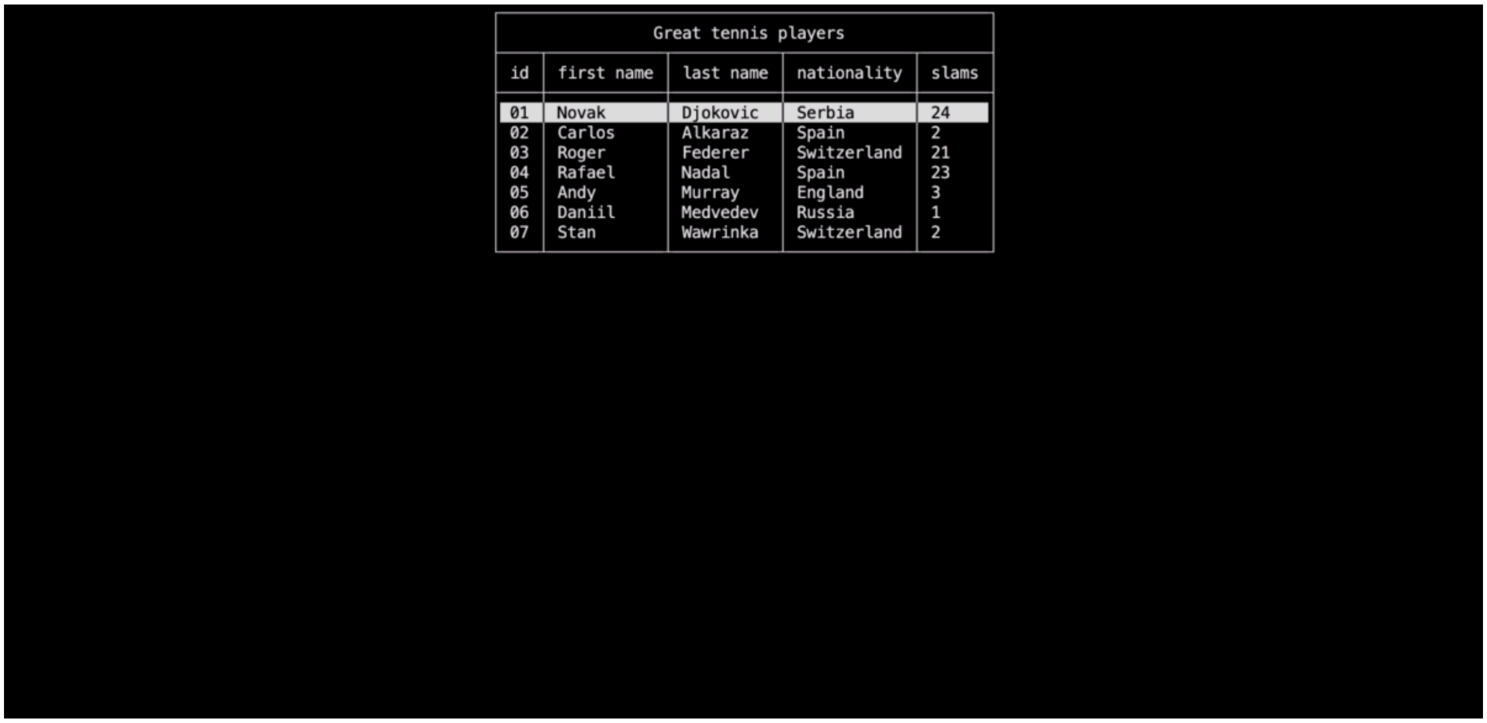
```
List<string> player4 = new List<string>() { "04", "Rafael", "Nadal", "Spain",
"23" };
List<string> player5 = new List<string>() { "05", "Andy", "Murray", "England",
"3" };
List<string> player6 = new List<string>() { "06", "Daniil", "Medvedev", "Russia",
"1" };
List<string> player7 = new List<string>() { "07", "Stan", "Wawrinka", "Switzerland",
"2" };

List<List<string>> playersData =
    new List<List<string>> ()
    {
        player1,
        player2,
        player3,
        player4,
        player5,
        player6,
        player7
    };

TableSelector players =
    new TableSelector(
        "Great tennis players",
         playersHeaders,
         playersData
    );

Window.AddElement(players);
// Contrary to the TableView, the TableSelector is interactive,
// so we do not have to stop the execution to see it, but to activate it
Window.ActivateElement(players);
```



```
                  Great tennis players
 id | first name | last name | nationality | slams
 01 | Novak      | Djokovic  | Serbia      | 24
 02 | Carlos     | Alkaraz   | Spain       | 2
 03 | Roger      | Federer   | Switzerland | 21
 04 | Rafael     | Nadal     | Spain       | 23
 05 | Andy       | Murray    | England     | 3
 06 | Daniil     | Medvedev  | Russia      | 1
 07 | Stan       | Wawrinka  | Switzerland | 2
```

Now let's collect the user interaction response by adding the following code:

```
var response = players.GetResponse();

Dialog playersEmbedResponse =
    new Dialog(
        new List<string>()
        {
            "Status: " + response!.Status,
            "You selected the player "
                + playersData[response!.Value][2]
                + "!"
        }
    );

Window.AddElement(playersEmbedResponse);
Window.ActivateElement(playersEmbedResponse);

Window.Close();
```

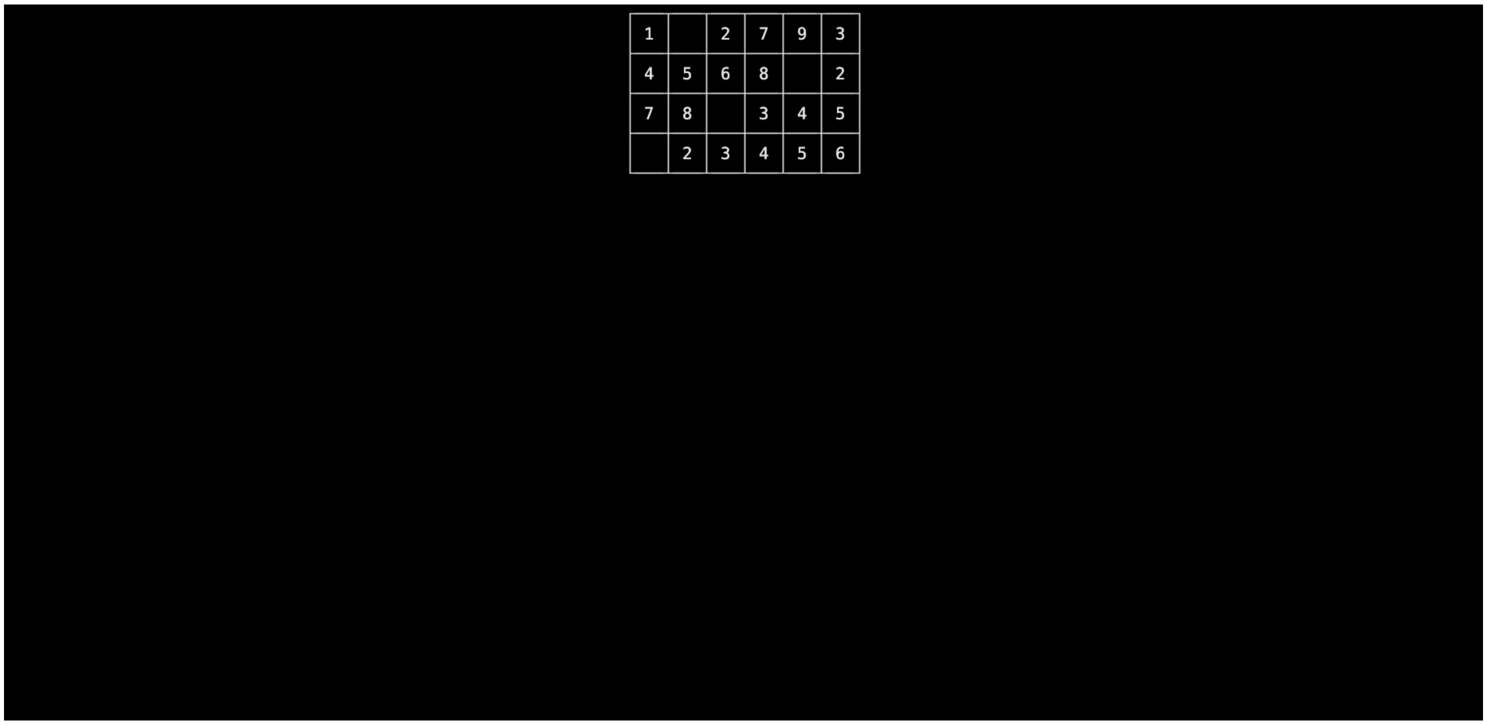


```
┌──────────────────────────────────────────────────────────┐
│                   Great tennis players                   │
├────┬─────────────┬───────────┬─────────────┬─────────────┤
│ id │ first name  │ last name │ nationality │ slams       │
├────┼─────────────┼───────────┼─────────────┼─────────────┤
│ 01 │ Novak       │ Djokovic  │ Serbia      │ 24          │
│ 02 │ Carlos      │ Alkaraz   │ Spain       │ 2           │
│ 03 │ Roger       │ Federer   │ Switzerland │ 21          │
│ 04 │ Rafael      │ Nadal     │ Spain       │ 23          │
│ 05 │ Andy        │ Murray    │ England     │ 3           │
│ 06 │ Daniil      │ Medvedev  │ Russia      │ 1           │
│ 07 │ Stan        │ Wawrinka  │ Switzerland │ 2           │
└────┴─────────────┴───────────┴─────────────┴─────────────┘
```

## The `Matrix` element

The `Matrix` element is used to display data in a matrix format. Learn more

```
Window.Open();

List<int?> firstRow = new List<int?>() { 1, null, 2, 7, 9, 3 };
List<int?> secondRow = new List<int?>() { 4, 5, 6, 8, null, 2 };
List<int?> thirdRow = new List<int?>() { 7, 8, null, 3, 4, 5 };
List<int?> fourthRow = new List<int?>() { null, 2, 3, 4, 5, 6 };

List<List<int?>> data =
    new List<List<int?>>() {
    firstRow,
    secondRow,
    thirdRow,
    fourthRow
};

Matrix<int?> matrix = new Matrix<int?>(data);

Window.AddElement(matrix);

Window.Render(matrix);
Window.Freeze();

Window.Close();
```



```
┌───┬───┬───┬───┬───┬───┐
│ 1 │   │ 2 │ 7 │ 9 │ 3 │
├───┼───┼───┼───┼───┼───┤
│ 4 │ 5 │ 6 │ 8 │   │ 2 │
├───┼───┼───┼───┼───┼───┤
│ 7 │ 8 │   │ 3 │ 4 │ 5 │
├───┼───┼───┼───┼───┼───┤
│   │ 2 │ 3 │ 4 │ 5 │ 6 │
└───┴───┴───┴───┴───┴───┘
```

## Conclusion

In this tutorial, you learned how to use the `TableView`, `TableSelector` and `Matrix` elements. You are now ready to start the menus management tutorial.

---

Have a question, give a feedback or found a bug? Feel free to [open an issue](#) or [start a discussion](#) on the GitHub repository.

# Menus management

In this section, you will learn:

- How to create a menu using the `ScrollingMenu` element
- Collect & manage their output

**TIP**

Each subsection is independent. I recommend you to overwrite the `Program.cs` file with the code of each section to avoid any confusion.

## The `ScrollingMenu` element

The `ScrollingMenu` *interactive* element is an historic element of the library. Some features about it changed over time but the principle has remain the same for a year. It is used to display a list of items and allow the user to select one or several items. Learn more

Here is a minimal example of how to use it:

```
Window.Open();

string[] options = new string[] { "Option 0", "Option 1", "Option 2" };
ScrollingMenu menu = new ScrollingMenu(
    "Please choose an option among those below.",
    0,
    Placement.TopCenter,
    options
);

Window.AddElement(menu);
// the ScrollingMenu is an interactive element, so we need to activate it
Window.ActivateElement(menu);

// The ScrollingMenu will return an int as a value (represents the index of the
selected item)
var responseMenu = menu.GetResponse();
Dialog embedResponse = new Dialog(
    new List<string>()
    {
        $"The user: {responseMenu!.Status}",
        $"Index: {responseMenu!.Value}",
        // We find the option selected by the user from the index
        $"Which corresponds to: {options[responseMenu!.Value]}"
    }
);

Window.AddElement(embedResponse);
Window.ActivateElement(embedResponse);

Window.Close();
```

## Manage menu status

The most practical way to manage actions according the the outcome of the `ScrollingMenu` is a switch-case statement. Learn more

Here is a basic example where we display a custom message according to the user's action (pressing Enter, Escape or Delete):

```
Window.Open();

string[] options = new string[] { "Option 0", "Option 1", "Option 2" };
ScrollingMenu menu = new ScrollingMenu(
    "Please choose an option among those below.",
    0,
    Placement.TopCenter,
    options
);
```

```
Window.AddElement(menu);
Window.ActivateElement(menu);

var response = menu.GetResponse();
switch (response!.Status)
{
    case Status.Selected:
        Dialog embedSelected = new Dialog(
            new List<string>()
            {
                "The user pressed the Enter key",
            }
        );
        Window.AddElement(embedSelected);
        Window.ActivateElement(embedSelected);

        Window.RemoveElement(embedSelected);
        break;
    case Status.Escaped:
        Dialog embedEscaped = new Dialog(
            new List<string>()
            {
                "The user pressed the Escape key",
            }
        );
        Window.AddElement(embedEscaped);
        Window.ActivateElement(embedEscaped);

        Window.RemoveElement(embedEscaped);
        break;
    case Status.Deleted:
        Dialog embedDeleted = new Dialog(
            new List<string>()
            {
                "The user pressed the Delete key",
            }
        );
        Window.AddElement(embedDeleted);
        Window.ActivateElement(embedDeleted);

        Window.RemoveElement(embedDeleted);
        break;
}

Window.Close();
```

## Manage menu value

As we mentioned earlier, the `ScrollingMenu` returns an `int` as a value. This value represents the index of the selected item. You may use it to act differently according to the selected item. Here we decide to act differently when the user selects an item and quit the app otherwise:

```
Window.Open();

string[] options = new string[] { "Play", "Settings", "Quit" };

ScrollingMenu menu = new ScrollingMenu(
    "Please choose an option among those below.",
    0,
    Placement.TopCenter,
    options
);

Window.AddElement(menu);
Window.ActivateElement(menu);

var response = menu.GetResponse();
switch (response!.Status)
{
    case Status.Selected:
        switch (response.Value)
        {
            case 0:
                Dialog play = new Dialog(
                    new List<string>() { "Playing..." }
                );
                Window.AddElement(play);
                Window.ActivateElement(play);

                Window.RemoveElement(play);
                break;
            case 1:
                Dialog settings = new Dialog(
                    new List<string>() { "Consulting the settings..." }
                );
                Window.AddElement(settings);
                Window.ActivateElement(settings);

                Window.RemoveElement(settings);
                break;
            case 2:
                // Quit the app
                Window.Close();
                break;
        }
        break;
    case Status.Escaped:
    case Status.Deleted:
        // Quit the app anyway
        Window.Close();
        break;
}
Window.Close();
```

That way, you may act differently depending on the selected item and create useful menu without too much effort.

## Conclusion

In this section, you learned how to create a menu using the `ScrollingMenu` element, collect and manage their output. Now let's jump to the final section!

---

Have a question, give a feedback or found a bug? Feel free to open an issue or start a discussion on the GitHub repository.

# Next steps

```
Well done!
You have finished the guided path through the general features of the library.
Starting from your first element, to a complex menu management to create real console applications.
You are now ready to create your own projects and to explore the advanced features of the library.
```

Now, you can go further and explore the library by yourself. Here are the resources available:

- The elements and references sections where you can find all the features available in the library.
- The examples section where you can find some concrete examples to help you understand how to use the library in real use cases.
- The articles section where you can find some additional articles (create your element, use fonts, create your documentation, etc.).

Stay tuned! The release notes of the latest version are updated regularly. Consider checking it to see if there are any new features or bug fixes to stay up to date. Updating instructions are available here.

If you have any questions, feel free to ask them in the discussions section or open an issue (templates are available to help you).

Finally, feel free to **share** around you and **star** the library on GitHub if you like the project!