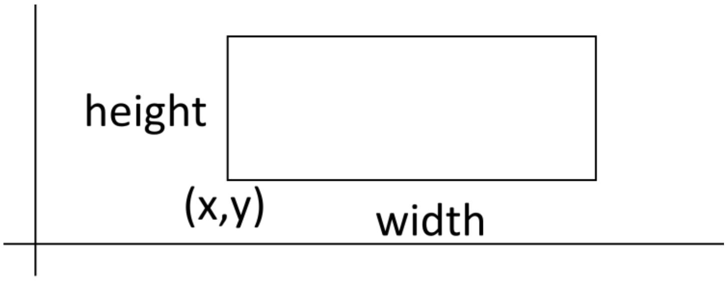
Exercises for Chapter 4 – The C# Classes

Exercise 1

- Create a class **Rectangle**.
  - The Rectangle object should have the following attributes:
    - X and Y coordinates of rectangle's bottom-left corner.
    - Width and Height
- Assumptions:
  - A rectangle object is parallel to the x,y coordinates.
  - The width attribute represents the side that is parallel to the **x** coordinate.
  - The height attribute represents the side that is parallel to the **y** coordinate.



- Attributes' values should be provided on a Rectangle object's initialization, and can be changed by resizing and moving the rectangle
- The Rectangle should provide the following properties and methods:
  - Get for its different attributes.
  - Area
    - Provides rectangle's area.
  - Resize()
    - Receives new width and height and resize the rectangle accordingly.
  - Move()
    - Receives new x and y coordinates and move the rectangle accordingly.
  - Assign()
    - Receives a rectangle as a parameter, and copies its attributes.
  - IsSizeEqual()

        - Receives a rectangle as parameter.
          Returns whether or not rectangles' area is equal.
    - GetUnion()
        - Receives a Rectangle object as parameter.
          Returns the minimum bounding rectangle.
    - Minimum()
        - Gets a rectangle as parameter.
          Returns the smaller of the two rectangles.
    - A print method that prints rectangle details to the console.
- In your **Main()** method (which should be defined in a separate class and file), do the following:
    - Get from the user an initial values for a rectangle attributes.
    - Print user rectangle details.
    - Create application rectangle (using hard coded initial values) and print its details.
    - Find out:
        - Is rectangles size equal?
        - Which is the minimal one?
        - Can you get their union?
    - Set your rectangle with the values of the user's one.
    - Try to compare them using the (==) operator.
        - Are they equal?
    - Assign user's rectangle into yours, using the (=) operator.
        - Are they equal (==) now?

Exercise 2

Implement a **`LinkedList`** class that contains strings.

- Advanced students may implement **`LinkedList<T>`**
- **`LinkedList`** should have the following methods:

```
Add(string item);
string GetAt(int index);
string RemoveAt(int index);
```

- **`LinkedList`** should have a **`Count`** property:
    - – Implement **`Count`** as an auto-property

- Implementation notes:
  - **LinkedList** should have a private inner class: **Node**
  - **Node** contains two elements: **Next** and **Item**
  - When you create a new **Node**, initialize it with object initializer
  - It is much easier to implement a cyclic list
    - Create a **private static readonly _emptyNode** node
    - Create a private **Node** reference (**_last**)
    - At the beginning:

      ```
       _last = _EmptyNode;
      _last.Next = EmptyNode;
      Count = 0;
      ```

    - Create a private **First** property that refer to the **_emptyNode.Next**
- At **Main()**, create a new list, and copy all command line arguments to the list.
- Test your **GetAt** and **Remove** method
  - To print the list after each change add a utility static class with **PrintList()** method, that get the list and print all elements
  - Call this method after each **Add** and **RemoveAt()** method call
  - Convert this method to be an extension method, call it **Print()**
- Add a **Log** partial method to the **LinkedList**

  ```
  static partial void Log(string message, T item, int count);
  ```

  - Remember to change the **LinkedList** to be partial class
  - Add calls to the **Log** method in **Add()** and **RemoveAt()**
- Create a second partial class and implement the **Log** method
  - Test your code
- The **ListUtil.Print()** extension method is very inefficient
  - Create an efficient **Print** method in the **LinkedList** class that override the extension method.