Contents

Properties

Introduction

- Attributes are implemented with a standard “template” of code
- Remember, “attribute” is the abstract concept of some data stored in an object; “instance variable” is the way that data is actually stored
- First, declare an instance variable for the attribute
- Then write a “getter” method for the instance variable
- Then write a “setter” method for the instance variable
- With this combination of instance variable and methods, the object has an attribute that can be read (with the getter) and written (with the setter)
- For example, this code implements a “width” attribute for the class Rectangle:

```
class Rectangle
{
    private int width;
    public void SetWidth(int value)
    {
        width = value;
    }
    public int GetWidth()
    {
        return width;
    }
}
```

- Note that there is a lot of repetitive or “obvious” code here:

  - The name of the attribute is intended to be "width," so you must name the instance variable `width`, and the methods `GetWidth` and `SetWidth`, repeating the name three times.
  - The attribute is intended to be type `int`, so you must ensure that the instance variable is type `int`, the getter has a return type of `int`, and the setter has a parameter type of `int`. Similarly, this repeats the data type three times.
  - You need to come up with a name for the setter's parameter, even though it also represents the width (i.e. the new value you want to assign to the width attribute). We usually end up naming it "widthParameter" or "widthParam" or "newWidth" or "newValue."

- Properties are a "shorthand" way of writing this code: They implement an attribute with less repetition.
- Note that properties are not present in every object-oriented programming language: for example, Java does not have properties[1].

Writing properties

- Declare an instance variable for the attribute, like before
- A **property declaration** has 3 parts:
  - Header, which gives the property a name and type (very similar to variable declaration)
  - `get` accessor, which declares the "getter" method for the property
  - `set` accessor, which declares the "setter" method for the property
- Example code, implementing the "width" attribute for Rectangle (this replaces the code in the previous example):

```
class Rectangle
{
    private int width;
    public int Width
    {
        get
        {
            return width;
        }
        set
        {
```

[1]https://stackoverflow.com/questions/2701077/does-java-have-properties-that-work-the-same-way-properties-work-in-c

```
            width = value;
        }
    }
}
```

- Header syntax: `[public/private] [type] [name]`
- *Convention* (not rule) is to give the property the same name as the instance variable, but capitalized – C# is case sensitive
- `get` accessor: Starts with the keyword `get`, then a method body inside a code block (between braces)
  - `get` is like a method header that always has the same name, and its other features are implied by the property's header
  - Access modifier: Same as the property header's, i.e. `public` in this example
  - Return type: Same as the property header's type, i.e. `int` in this example (so imagine it says `public int get()`)
  - Body of `get` section is exactly the same as body of a "getter": return the instance variable
- `set` accessor: Starts with the keyword `set`, then a method body inside a code block
  - Also a method header with a fixed name, access modifier, return type, and parameter
  - Access modifier: Same as the property header's, i.e. `public` in this example
  - Return type: Always `void` (like a setter)
  - Parameter: Same type as the property header's type, name is always "value". In this case that means the parameter is `int value`; imagine the method header says `public void set(int value)`
  - Body of `set` section looks just like the body of a setter: Assign the parameter to the instance variable (and the parameter is always named "value"). In this case, that means `width = value`

Using properties

- Properties are members of an object, just like instance variables and methods
- Access them with the "member access" operator, aka the dot operator
  - For example, `myRect.Width` will access the property we wrote, assuming `myRect` is a Rectangle

- A complete example (available as a complete solution[2]), where the "length" and "width" attributes are implemented with properties:

  ```
  class Rectangle
  {
    private int width;
    public int Width
    {
      get { return width; }
      set { width = value; }
    }
    private int length;
    public int Length
    {
      get { return length; }
      set { length = value; }
    }
  }
  ```

- Properties "act like" variables: you can assign to them and read from them

- Reading from a property will *automatically* call the `get` accessor for that property

  - For example,

  ```
  Console.WriteLine(
      $"The width is {myRectangle.Width}");
  ```

  will call the `get` accessor inside the `Width` property, which in turn executes `return width` and returns the current value of the instance variable

  - This is equivalent to

  ```
  Console.WriteLine(
      $"The width is {myRectangle.GetWidth()}");
  ```

  using the "old" Rectangle code

- Assigning to (writing) a property will *automatically* call the `set` accessor for that property, with an argument equal to the right side of the = operator

  - For example, `myRectangle.Width = 15;` will call the `set` accessor inside the `Width` property, with `value` equal to 15

[2]https:/princomp.github.io/code/projects/Properties_Example.zip

  - This is equivalent to `myRectangle.SetWidth(15);` using the "old" Rectangle code

In More Details

- Note that in a property, `value` is what is called a *contextual keyword*[3]: it is not a reserved word in C# (it could be used as an identifier), but *inside a property* it refers to something special, the name of the set method parameter.
- In the following code:

  ```
  private int width;
  public int Width
  {
      get
      {
          return width;
      }
      set
      {
          width = value;
      }
  }
  ```

  The attribute `width` is called *the Width's property backing field*: it holds the data assigned to the property.
- When the property's get *and* set accessors are trivial (like the ones above), we can simply omit them their body completely. That is, the previous `Width` property could be implemented using

  ```
  public int Width { get; set;  }
  ```

  This is called *auto-properties*. Note that in this case, we do not need to declare the property's backing field (that is, no need to have `private int width;`), but cannot refer to it!
- Conversely, get and set accessor can contains arbitrarily convoluted code:

  ```
  public int Length
  {
      get { return length; }
      set {
          if (value < 0) {
              length = -value;
  ```

[3]https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/lexical-structure#644-keywords

```
            }
            else if (value == 0) {
                length = 1;
            }
            else {
                length = value;
            }
        }
    }
```

- Note however that if the set or get accessor is not the "trivial" one, then auto-properties cannot be used and the other accessor must be specified.
  - For example, in the above code, simply writing `get;` instead of `get { return length; }` would give a compilation error.
- Note that properties can exist without backing field, and they can be *read-only* (that is, without a set accessor) or *write-only* (that is, without a get accessor, but this is rarer).
  - An example of read-only property is as follows:

    ```
    class Circle
    {
        public decimal Diameter { get; set; }
        // The constructor below  sets the value
        // of the property's backing field through
        // the property's set accessor.
        public Circle(decimal dP)
        {
            Diameter = dP;
        }
        // The Radius property below is
        // 1. read-only (no set accessor),
        // 2. without a backing field.
        public decimal Radius {
            get { return Diameter / 2; }
        }
    }
    ```
- It is possible to set a "custom default value" for properties using a *property initializer*, as follows:

  ```
  public double Width { get; set; } = -1;
  ```

  In this case, the property's backing field value will be -1 by default. Properties with initializer can be read-only:

  ```
  public int MaximumValue { get; } = 999;
  ```

- Finally, properties can be **static** as well:

  ```
  public static string Explanation { get; set; } = "A
   ↪  Circle has for radius its diameter divided by 2.";
  ```

  Such a property can be accessed using for example

  ```
  Console.WriteLine(Circle.Explanation);
  ```

  and its value can be changed, for instance by appending a `string` to it:

  ```
  Circle.Explanation += "\nIts circumference is π
   ↪  multiplied by its diameter.";
  ```

Properties in UML Class Diagrams

Simple Notation

- Since properties represent (or, rather, allow to access) attributes, they go in the "attributes" box (the second box)
- If a property will simply "get" and "set" an instance variable of the same name, you do *not* need to write the instance variable in the box
  - No need to write both the property `Width` and the instance variable `width`
- Syntax: `[+/-] <<property>> [name]: [type]`
- Note that the access modifier (+ or -) is for the property, not the instance variable, so it is + if the property is **`public`** (which it usually is)
- Example for `Rectangle`, assuming we converted both attributes to use properties instead of getters and setters:
- We no longer need to write all those setter and getter methods, since they are "built in" to the properties

More Accurate Notation In general, instead of writing for example

```
+ <<properties>> Explanation: string
```

one can write

```
+ <<get, set>> Explanation: string
```

or even

```
+ <<set>> Explanation: string
+ <<get>> Explanation: string
```

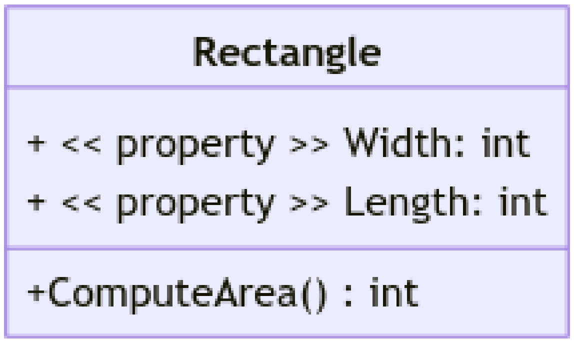

Figure 1: A UML diagram for the Rectangle class (text version[4])

The benefit of this notation is that read-only properties can easily be integrated in the UML class diagram, by simply omitting the <<`set`>> line:

```
+ <<get>> Radius : decimal
```