Getters, Setters and Properties

In this lesson, we will look into the getters, setters and properties in C#.

WE'LL COVER THE FOLLOWING

An Analogy

Consider the vending machine example. In this machine, there are several components working together at a micro level to make the vending machine functional at a macro level. There may be a display screen whose purpose is to provide an interface to the users. There may be a product dispenser whose purpose is to dispense products. There may be a cash collector whose purpose is to collect cash from the customers.

What we want to emphasize here is that there are many components providing their specific functionalities that don't know each other's internal implementation or working details. The screen doesn't need to know how the cash collector component is internally keeping track of the total money collected until now, nor does the cash collector need to know how the screen is controlling its brightness.

While designing classes in OOP we follow a similar technique. Our classes should hide their internal implementation details and logic from the other classes. This is a very important concept called *Encapsulation* and we are going to explore it in detail in the upcoming chapter.

We already know that we declare fields in our classes. These fields actually represent the implementation details of our classes. In order to hide these from other classes, we need to declare them as `private`. The question here is

"If all the fields should be `private` then how can we access and use these in our code in `Main()` or some other class?"

Getters and Setters

Getter and setter methods are the answer to the above question.

These two types of methods are very popular in OOP. A **get** method retrieves the value of a particular field, whereas a **set** method changes its value.

> Get and set methods are implemented for fields that are declared `private`. It is considered a good practice to declare all the fields private.

While writing a getter or a setter method, it is a common convention to write the name of the corresponding field preceded by the *Get* or *Set* words, respectively.

Let's write getter and setter methods for the `count` field in our `VendingMachine` class:

```
// VendingMachine class
class VendingMachine {

  private int _count; // member field count
  private int _capacity = 100;

  // Setter method to set the count of the products
  public void SetCount(int x) {
    if(x >=0 && x <= _capacity) { // count should always be positive and less than or equal t
    _count = x;
    }
  }

  // Getter method to get the count of the products
  public int GetCount() {
    return _count;
  }

}

class Demo {

   public static void Main(string[] args) {
     var vendingMachine = new VendingMachine();
     vendingMachine.SetCount(88); // calling the setter method
     Console.WriteLine("The count is: {0}", vendingMachine.GetCount()); // calling the getter
    }
}
```

We defined the getter and setter methods for the `private` field `count` in the above code. At **line 8** we notice that certain conditional statements can also be checked while setting or accessing the value using setters and getters. Allowing arbitrary public access to a field can result in all sorts of runtime errors. Setters enable controlled access to `private` fields.

Properties

Note the declaring fields as private and then implementing the getter and setter methods is a common case. C# provides an alternative syntax called "Properties", which makes it easier to implement controlled public access to private state.

> **Properties** are class members which can be used as an alternative to the getter and setter methods.

In C#, we can implement properties in our class to achieve the functionality of getters and setters with a lesser piece of code. The generic signature of a property is as follows:

```
public DataTypeOfField PropertyName {
  get{...}
  set{...}
}
```

Let's have a look at it:

```
// VendingMachine class
class VendingMachine {

  private int _count = 0; // member field count
  private int _capacity = 100;

  // Implementing the count property
  public int Count {
    get { //for getting value
```

```
        return _count;
        }
      set { //for setting value

        if (value >= 0 && value <= _capacity)
        _count = value;
        }
    }

}

class Demo {

   public static void Main(string[] args) {
     var vendingMachine = new VendingMachine();
     vendingMachine.Count = 88; // setting the count using Count property
     Console.WriteLine("The count is: {0}", vendingMachine.Count); // getting the count using
   }

}
```

We can see how easily we can get and set the values of a `private` field by implementing a property. While setting the value of the field we have used the keyword `value`.

> If we want a read-only or write-only field, we have the choice to define the `get` block only inside a property by skipping the definition of `set` block and vice versa.

Just like getter and setter methods we can also use conditional statements while implementing a property. Another thing to note here is that while accessing a property in `Main()` we have used the dot, `.`, operator.

Auto-Implemented Properties

What if someone tells us that we can shorten the code of properties to achieve similar functionality to that of getter and setter methods? That would be awesome, wouldn't it?

C# provides us the option of using auto-implemented properties. In this case, we don't even have to declare the field separately. Let's have a look into it:

```
class VendingMachine {
  //no fields declared
  // Implementing the auto-implemented property for count
  public int Count {get; set;}


}

class Demo {

   public static void Main(string[] args) {
     var vendingMachine = new VendingMachine();
     vendingMachine.Count = 88; // setting the count using Count property
     Console.WriteLine("The count is: {0}", vendingMachine.Count); // getting the count using
   }

}
```

In the above code, we can notice how efficiently, on **line 5**, we have done everything using the auto-implemented property. This property is called auto-implemented because when we use it, the compiler itself declares a `private` field and implements its getter and setter for us.

> **Exercise caution:** Use auto-implemented properties only when you are absolutely certain that there are no safety or code correctness issues.

One drawback of this auto-implemented property is that we cannot set any conditional checks while setting or accessing the fields.

This ends our lesson about the getters, setters, and properties. In the next lesson, we will learn how to create objects in a more efficient and usable way.