Lesson 24 - Classes & Objects V

Indexers

An **indexer** allows objects to be indexed like an array. A string variable is actually an object of the String class. Further, the String class is actually an array of Char objects. In this way, the string class implements an indexer so we can access any character (Char object) by its index:

```
string str = "Hello World";
char x = str[4];
Console.WriteLine(x);
```

> Arrays use integer indexes, but indexers can use any type of index (strings, characters, etc.)

Declaration of an indexer is to some extent similar to a property. The difference is that indexer accessors require an **index**. Like a property, you use `get` and `set` accessors for defining an indexer. However, where properties return or set a specific data member, indexers return or set a particular value from the object instance. Indexers are defined with the `this` keyword.

```
class Clients {
  private string[] names = new string[10];

  public string this[int index] {
    get {
      return names[index];
    }
    set {
      names[index] = value;
    }
  }
}
```

As you can see, the indexer definition includes the `this` keyword and an index, which is used to get and set the appropriate value. Now, when we declare an object of class Clients, we use an index to refer to specific objects like the elements of an array:

```
Clients c = new Clients();
c[0] = "Talha";
c[1] = "Amad";

Console.WriteLine(c[1]);
```

> You typically use an indexer if the class represents a list, collection, or array of objects.

Operator Overloading

Most operators in C# can be overloaded, meaning they can be redefined for custom actions. For example, you can redefine the action of the plus (+) operator in a custom class. Consider the Box class that has Height and Width properties:

```
class Box {
  public int Height {get; set;}
  public int Width {get; set;}
  public Box(int h, int w) {
    Height = h;
    Width = w;
  }
}
static void Main(string[] args) {
  Box b1 = new Box(14, 3);
  Box b2 = new Box(5, 7);
}
```

We would like to add these two Box objects, which would result in a new, bigger Box. So, basically, we would like the following code to work:

```
 Box b3 = b1 + b2;
```

The Height and Width properties of object b3 should be equal to the sum of the corresponding properties of the b1 and b2 objects.

> This is achieved through operator overloading.

Overloaded operators are methods with special names, where the keyword `operator` is followed by the symbol for the operator being defined. Similar to any other method, an overloaded operator has a return type and a parameter list. For example, for our `Box` class, we overload the + operator:

```
public static Box operator+ (Box a, Box b) {
  int h = a.Height + b.Height;
  int w = a.Width + b.Width;
  Box res = new Box(h, w);
```

```
  return res;
}
```

The method above defines an overloaded `operator +` with two Box object parameters and returning a new Box object whose Height and Width properties equal the sum of its parameter's corresponding properties. Additionally, the overloaded operator must be `static`. Putting it all together:

```
class Box {
        public int Height { get; set; }
        public int Width { get; set; }
        public Box(int h, int w) {
                Height = h;
                Width = w;
        }
        public static Box operator+(Box a, Box b) {
                int h = a.Height + b.Height;
                int w = a.Width + b.Width;
                Box res = new Box(h, w);
                return res;
        }
}
```

```
Box b1 = new Box(14, 3);
Box b2 = new Box(5, 7);
Box b3 = b1 + b2;

Console.WriteLine(b3.Height);
Console.WriteLine(b3.Width);
```

> All arithmetic and comparison operators can be overloaded. For instance, you could define greater than and less than operators for the boxes that would compare the Boxes and return a **boolean** result. Just keep in mind that when overloading the greater than operator, the less than operator should also be defined.