Object Oriented Programing in C#

So far

Methods: Behavior

Variables: Data (state)

Classes

Combine Behavior and State

```
using System;

namespace EmployeeDatabase
{
  class Program
  {

    static void DisplayGreeting()
    {
      Console.WriteLine("----------------------------------------");
      Console.WriteLine("    Welcome to Our Employee Database    ");
      Console.WriteLine("----------------------------------------");
      Console.WriteLine();
      Console.WriteLine();
    }

    static string PromptForString(string prompt)
    {
      Console.Write(prompt);
      var userInput = Console.ReadLine();

      return userInput;
    }

    static int PromptForInteger(string prompt)
    {
      Console.Write(prompt);
      int userInput;
      var isThisGoodInput = Int32.TryParse(Console.ReadLine(), out userInput);

      if (isThisGoodInput)
      {
        return userInput;
      }
      else
      {
        Console.WriteLine("Sorry, that isn't a valid input, I'm using 0 as your answer.");
        return 0;
      }
    }

    static int ComputeMonthlySalaryFromYearly(int yearlySalary)
    {
      return yearlySalary / 12;
    }

    static void Main(string[] args)
    {
      DisplayGreeting();

      var name = PromptForString("What is your name? ");

      int department = PromptForInteger("What is your department number? ");

      int salary = PromptForInteger("What is your yearly salary (in dollars)? ");

      int monthlySalary = ComputeMonthlySalaryFromYearly(salary);

      Console.WriteLine($"Hello, {name} you make {monthlySalary} dollars per month.");
    }
  }
}
```

What if we wanted to add information about a second employee? Certainly one approach would be to add a second set of variables such as:

```
var name1 = PromptForString("What is your name? ");
int department1 = PromptForInteger("What is your department number? ");
int salary1 = PromptForInteger("What is your yearly salary (in dollars)? ");
int monthlySalary1 = ComputeMonthlySalaryFromYearly(salary1);

var name2 = PromptForString("What is your name? ");
int department2 = PromptForInteger("What is your department number? ");
int salary2 = PromptForInteger("What is your yearly salary (in dollars)? ");
int monthlySalary2 = ComputeMonthlySalaryFromYearly(salary2);
```

Or perhaps we would have some arrays

- `names[]`
- `departments[]`

This has some drawbacks. We'd have to know that `names[0]` is related to `departments[0]`, etc.

If we removed something from `names` we'd have to be sure to remove the corresponding `departments` entry.

Put related data in the same state

To group related data we consider them all part of the same `state`.

In this case the `state` is about an `Employee` who has individual attributes:

- `name`
- `department`
- `salary`
- and something we compute named `monthlySalary`

A diagram of this might look like:

```
+-------------------+
|     Employee      |
+-------------------+
|   name            |
|   department      |
|   salary          |
|   monthlySalary   |
+-------------------+
```

If we had a few of these `Employee` things around we might also see they could have their own, specific values for each of these attributes.

Employee	
ATTRIBUTE	VALUE
name	Elon Musk
department	42
salary	120000
monthlySalary	10000

Employee	
ATTRIBUTE	VALUE
name	Grace Hopper
department	100
salary	240000
monthlySalary	20000

Grouping these attributes together: Encapsulation

Keep track of this data together!

Class Syntax

Let's take our idea of an *employee* in this system and write some syntax to help us.

```
class
```

```
//
// class keyword
// |
// |    Name of class (PascalCase)
// |    |
// v    v
  class Employee
  {
    // public means "this can be seen outside of the class"
    // |
    // |   Type
    // |   |
    // |   |      Name of property
    // |   |      |
    // |   |      |
    // |   |      |
    // v   v      v
    public string Name;
    public int Department;
    public int Salary;
    public int MonthlySalary;

    // This is a *special* method known as a "constructor"
    // The constructor is called when we write a line like: `var bob = new Employee(`
    // The arguments to the method should line up to those below
    //
    //              This will become the employee's name
    //              |               This will become the employee's department
    //              |               |                  This will become the employee's salary
    //              |               |                  |              This will become the employee's monthly salary
    //              |               |                  |              |
    //              v               v                  v              v
    public Employee(string newName, int newDepartment, int newSalary, int newMonthlySalary)
    {
        // In the constructor we should setup the values for any of the properties.
        // Here we will *copy* the values given by the arguments to the corresponding property.

        Name = newName;
        Department = newDepartment;
        Salary = newSalary;
        MonthlySalary = newMonthlySalary;
    }
  }
```

Properties

These represent the DATA part of state and data.

Accessible both inside the class and outside.

```
// public means "this can be seen outside of the class"
// |   Type
// |   |      Name of property
// v   v      v
public string Name;
public int Department;
public int Salary;
public int MonthlySalary;
```

Constructor

- Special method.
- Used when we create an *instance* of an object
- May include arguments used to initialize properties

```
//                New Name
//                |                New Department
//                |                |                   New Salary
//                |                |                   |               New Monthly Salary
//                |                |                   |               |
//                v                v                   v               v
public Employee(string newName, int newDepartment, int newSalary, int newMonthlySalary)
```

Constructor

- Responsible for setting up default values of properties
- Anything else that happens at *make a new object* time
- In our case, take each of the arguments and *copy* them into the properties

```
{
  Name = newName;
  Department = newDepartment;
  Salary = newSalary;
  MonthlySalary = newMonthlySalary;
}
```

Example usage

We refer to this as *instantiating objects*

Notice this has a similar syntax to `new List`

We've already been making objects!

```
var graceHopper = new Employee("Grace Hopper", 100, 240000, 20000);
Console.WriteLine(graceHopper.Department); // Will show 100

var elonMusk = new Employee("Elon Musk", 42, 120000, 10000);
Console.WriteLine(elonMusk.Department); // Will show 42
```

We make objects like this *often*

- C# gives us a shortcut to create a new object and fill in properties
- `getters and setters`
- `defaults`

Properties with *getters* and *setters*

```
//
// class keyword
// |
// |    Name of class (PascalCase)
// |    |
// v    v
  class Employee
  {
    // public means "this can be seen outside of the class"
    // |
    // |   Type
    // |   |
    // |   |      Name of property
    // |   |      |
    // |   |      |      We can get the data and set the data
    // |   |      |      |
    // v   v      v      v
    public string Name { get; set; }
    public int Department { get; set; }
    public int Salary { get; set; }
    public int MonthlySalary { get; set; }
  }
```

We can *instantiate* employees.

```
+-----------------------------+
|      Employee Object        |
+---------------+-------------+
| name          | Elon Musk   |
| department    | 42          |
| salary        | 120000      |
| monthlySalary | 10000       |
+---------------+-------------+
```

```
+-----------------------------+
|        Employee Object      |
+---------------+-------------+
| name          | Grace Hopper |
| department    | 100         |
| salary        | 240000      |
| monthlySalary | 20000       |
+---------------+-------------+
```

```
var firstEmployee = new Employee();
var secondEmployee = new Employee();
```

Accessing properties

```
var firstEmployee = new Employee();
var secondEmployee = new Employee();

firstEmployee.Name = "Elon Musk";
secondEmployee.Name = "Grace Hopper";
```

What do our instances look like?

```
+-----------------------------+
|      firstEmployee          |
+---------------+-------------+
| Name          | "Elon Musk" |
| Department    | 0           |
| Salary        | 0           |
| MonthlySalary | 0           |
+---------------+-------------+
```

```
+-------------------------------+
|        secondEmployee         |
+---------------+---------------+
| Name          | "Grace Hopper" |
| Department    | 0              |
| Salary        | 0              |
| MonthlySalary | 0              |
+---------------+---------------=
```

Let's fill in the rest of the properties.

```
var firstEmployee = new Employee();
var secondEmployee = new Employee();

firstEmployee.Name = "Elon Musk";
firstEmployee.Department = 42;
firstEmployee.Salary = 120000;
firstEmployee.MonthlySalary = 10000;

secondEmployee.Name = "Grace Hopper";
secondEmployee.Department = 100;
secondEmployee.Salary = 240000;
secondEmployee.MonthlySalary = 20000;
```

Now our instances look like this:

```
+-----------------------------+
|       firstEmployee         |
+---------------+-------------+
| Name          | "Elon Musk" |
| Department    | 42          |
| Salary        | 120000      |
| MonthlySalary | 10000       |
+---------------+-------------+
```

```
+-------------------------------+
|        secondEmployee         |
+---------------+---------------+
| Name          | "Grace Hopper" |
| Department    | 100           |
| Salary        | 240000        |
| MonthlySalary | 20000         |
+---------------+---------------+
```

Simpler syntax

When creating a new object, C# gives us a convenient syntax:

```
var firstEmployee = new Employee {
  Name = "Elon Musk",
  Department = 42,
  Salary = 120000,
  MonthlySalary = 10000
};

var secondEmployee = new Employee {
  Name = "Grace Hopper",
  Department = 100,
  Salary = 240000,
  MonthlySalary = 20000
};
```

Update our code

So far we have seen how `class-es:`

- Store data, which we call `state`, in attributes we call `properties`.
- Are the template that describes what data is used
- Create *instances*, called `objects`
- Are like cookie cutters, where `objects` are like the cookies

Behavior with methods

The code has a method `ComputeMonthlySalaryFromYearly` that accesses our object's property and does some math.

We also have a *property* named `MonthlySalary`.

Any time we change the `Salary` we should be able to ask for the `MonthlySalary`.

Make `MonthlySalary` a method

```
class Employee
{
  public string Name { get; set; }
  public int Department { get; set; }
  public int Salary { get; set; }
  public int MonthlySalary()
  {
    return Salary / 12;
  }
}
```

Create the following objects

```
var firstEmployee = new Employee {
  Name = "Elon Musk",
  Department = 42,
  Salary = 120000,
};

var secondEmployee = new Employee {
  Name = "Grace Hopper",
  Department = 100,
  Salary = 240000,
};
```

```
+-----------------------------+
|      firstEmployee          |
+---------------+-------------+
| Name          | "Elon Musk" |
| Department    | 42          |
| Salary        | 120000      |
| MonthlySalary | METHOD      |
+---------------+-------------+
```

```
+-------------------------------+
|        secondEmployee         |
+---------------+---------------+
| Name          | "Grace Hopper" |
| Department    | 100           |
| Salary        | 240000        |
| MonthlySalary | METHOD        |
+---------------+---------------+
```

Objects have their own data (state) but shared behavior (methods)

For `firstEmployee.MonthlySalary()`

- Runs `return Salary / 12`
- But `C#` knows we are the instance `firstEmployee`
- Thus `Salary` is `120000` and we get back `10000`.

Objects have their own data (state) but shared behavior (methods)

For `secondEmployee.MonthlySalary()`

- Runs `return Salary / 12`
- But `C#` knows we are the instance `secondEmployee`
- Thus `Salary` is `240000` and we get back `20000`.

Reusability

Inheritance

Often in modeling the real world we encounter different types of data that are related. Sometimes these are in an *is a* or *is a kind of* relationship.

For instance in our system perhaps we have active employees and retired employees. In this case perhaps we want to return a `0` for the `MonthlySalary` of any retired employees.

```
//
//    This class
//     |
//     |                Is a kind of this class
//     |                |
//     |                |
//     v                v
class RetiredEmployee : Employee
{
    public int MonthlySalary()
    {
      return 0;
    }
}
```

Now if we defined a new employee as such:

```
var thirdEmployee = new RetiredEmployee {
    Name = "Bill Gates",
    Department = 100,
    Salary = 120,
};

Console.WriteLine(thirdEmployee.MonthlySalary());
```

This would output `0`. Notice we did not have to redeclare `Name`, or `Department`, or `Salary` as those are **inherited** from the base `class` of `Employee`.

Inheriting allows us to both *add* new state and behavior and *override* behavior from the base class.

Inheritance is a powerful tool but it is used less often in favor of other techniques such as

- *extensions*
- *mixins*
- *dependency injection*

We will discuss some of these in other lessons.