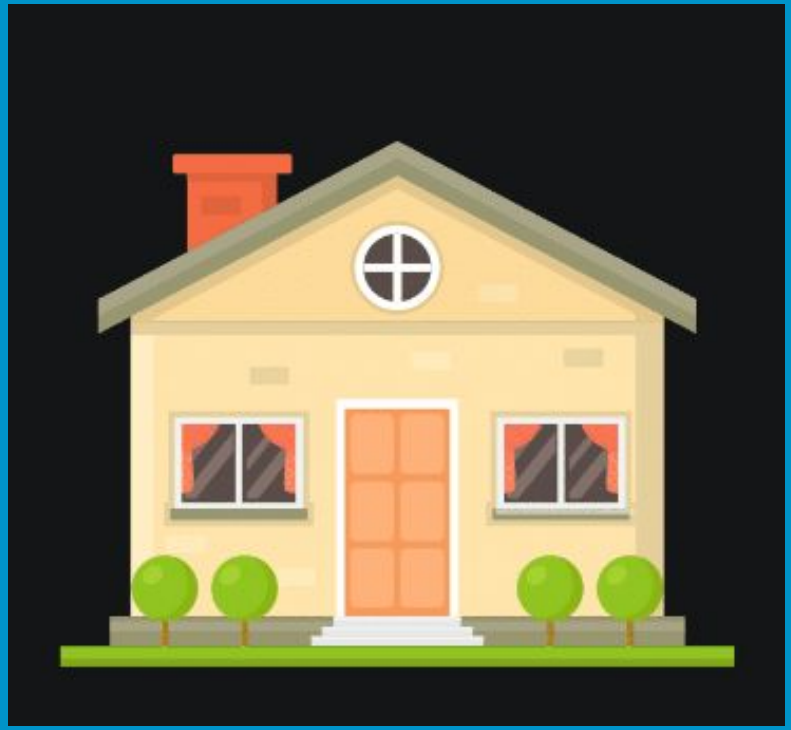
Classes




static keyword

Using classes will allow us to implement:

OOPs A PIE

Stands for:

A - abstraction
P - polymorphism
I - inheritance
E - encapsulation

OOPs A PIE

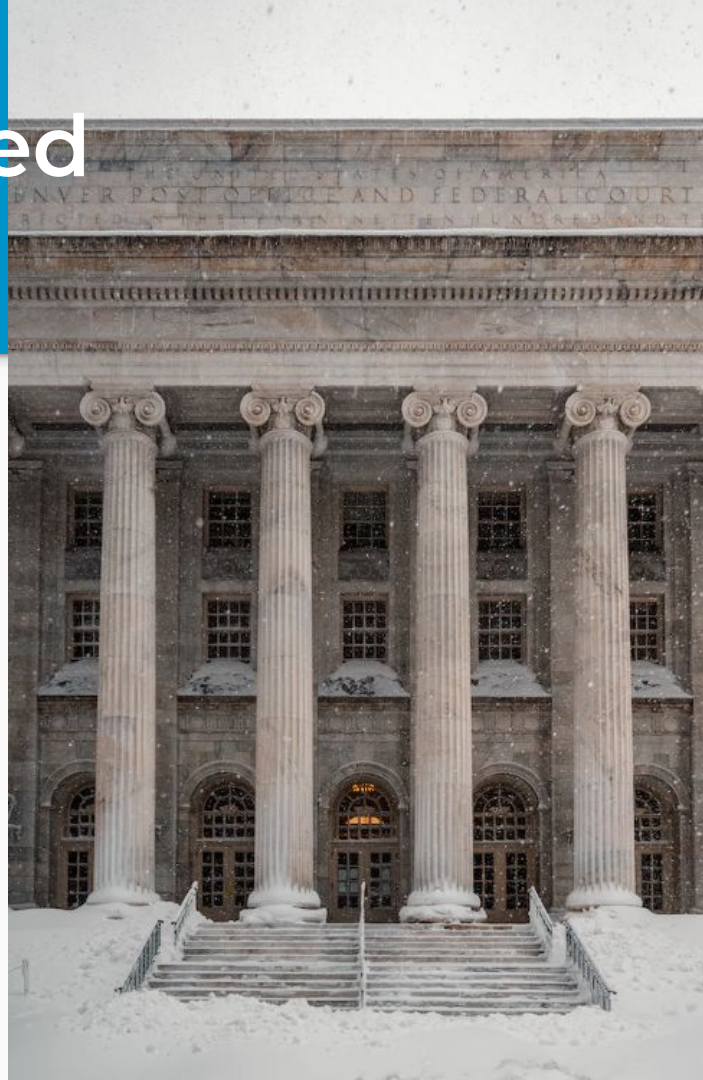
Stands for:

A - abstraction
P - polymorphism
I - inheritance
E - encapsulation

4 Pillars of Object Oriented Programming

1. Abstraction
2. Polymorphism
3. Inheritance
4. Encapsulation

*These four pillars revolve around classes!



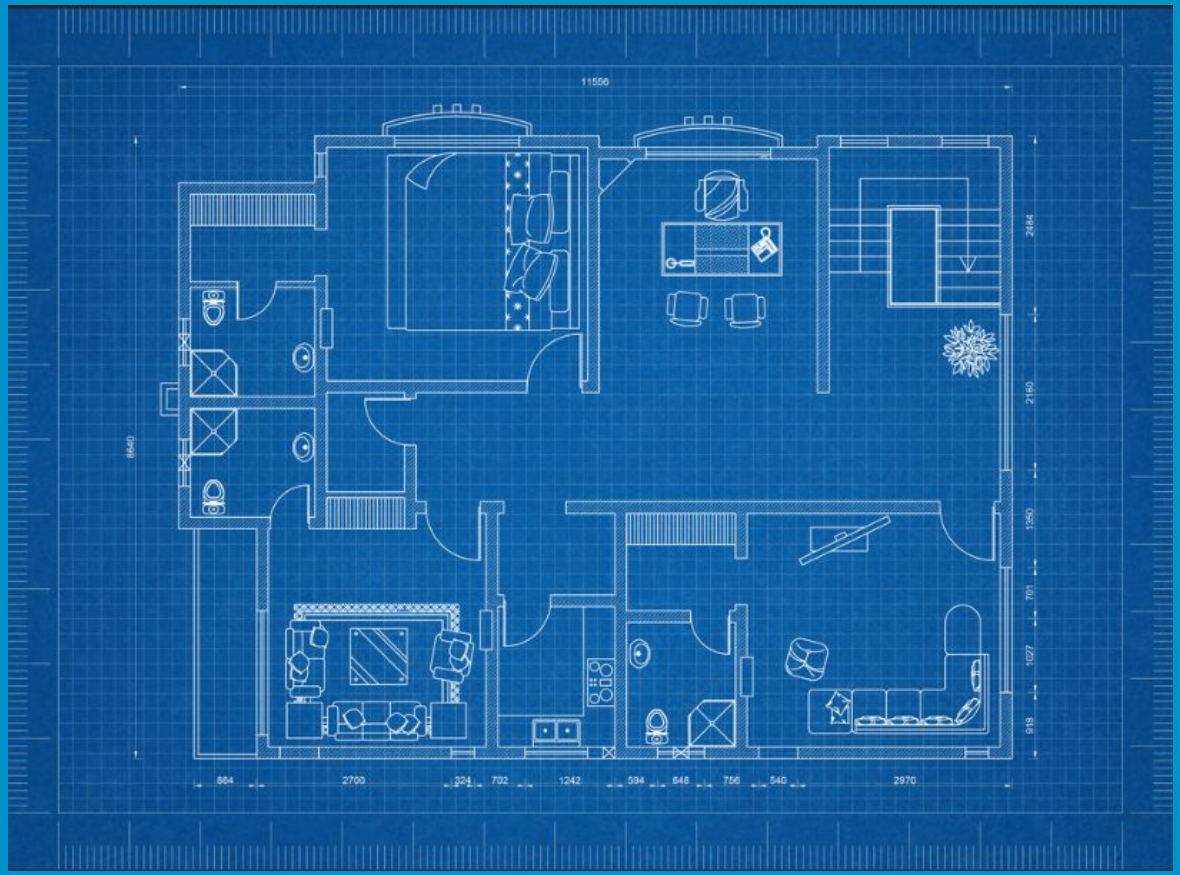
Classes

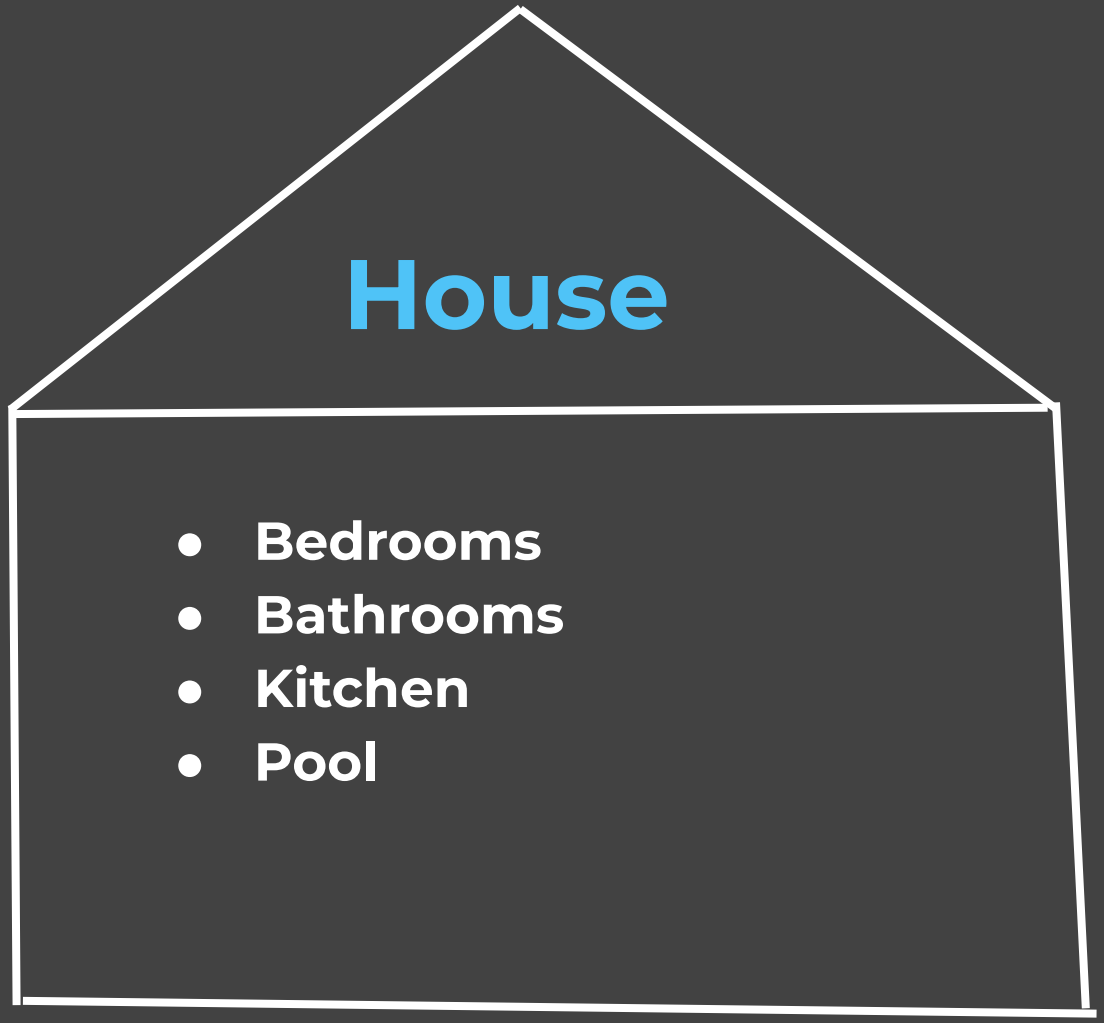
-Classes are the **cornerstone** of object oriented programming

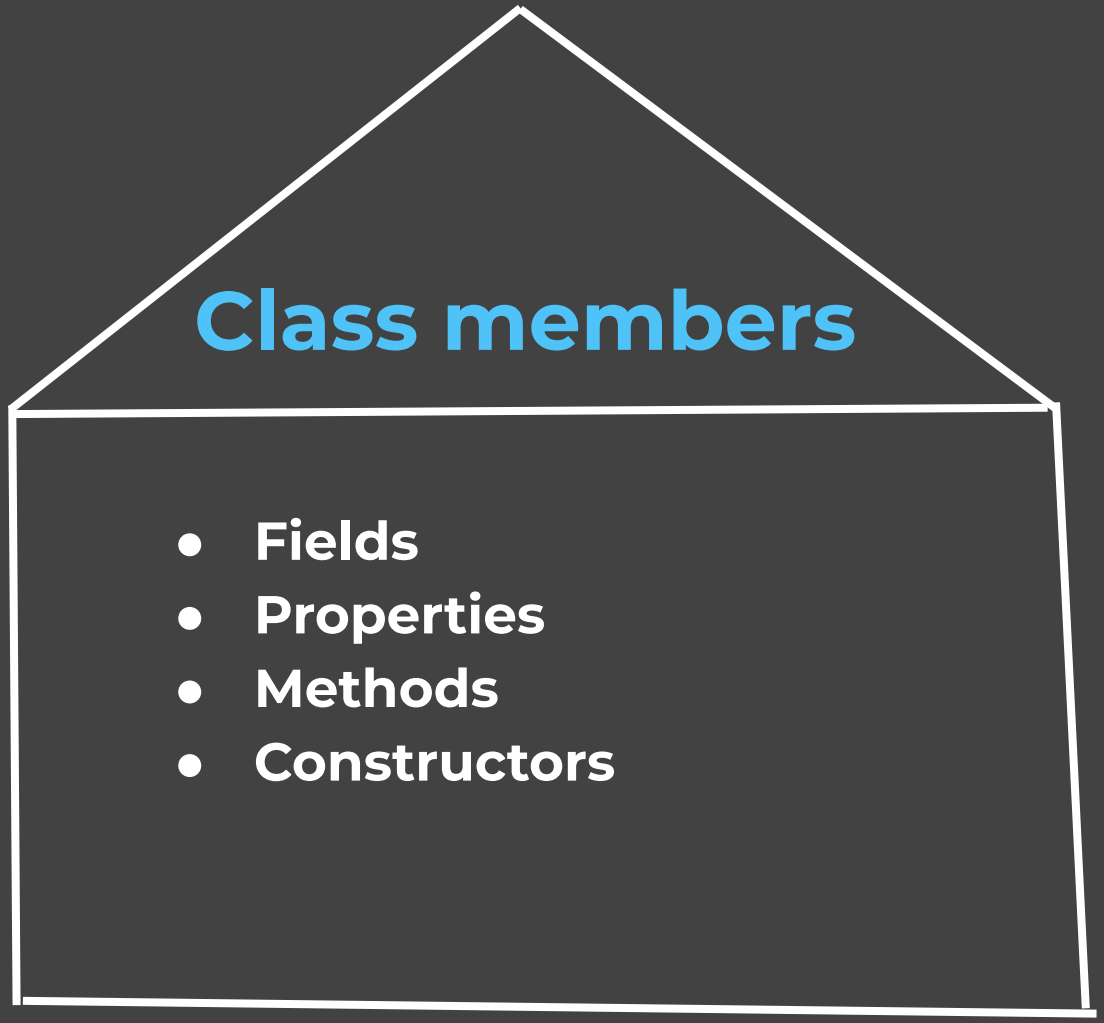
Classes

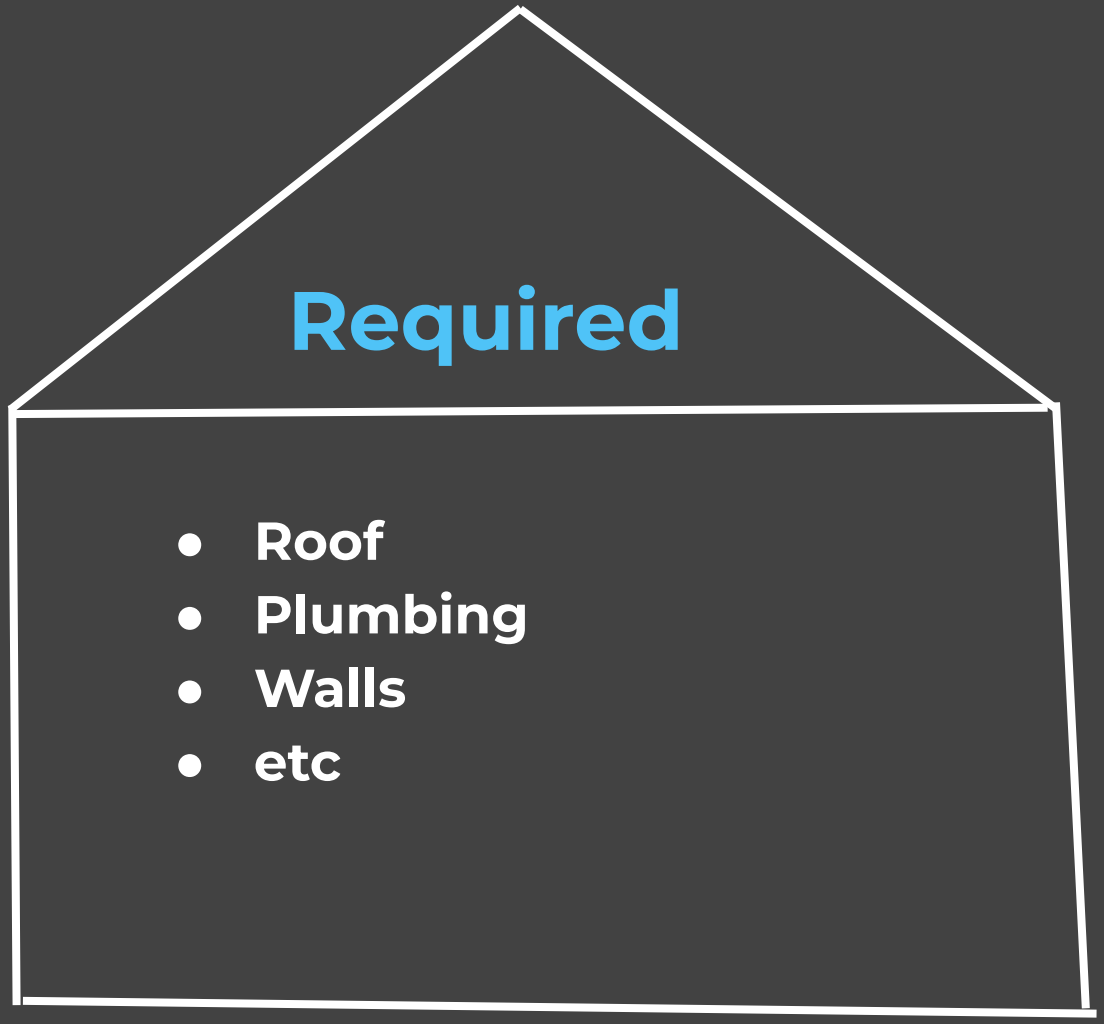
- Allow us to create objects
- C# is an OBJECT oriented programming language

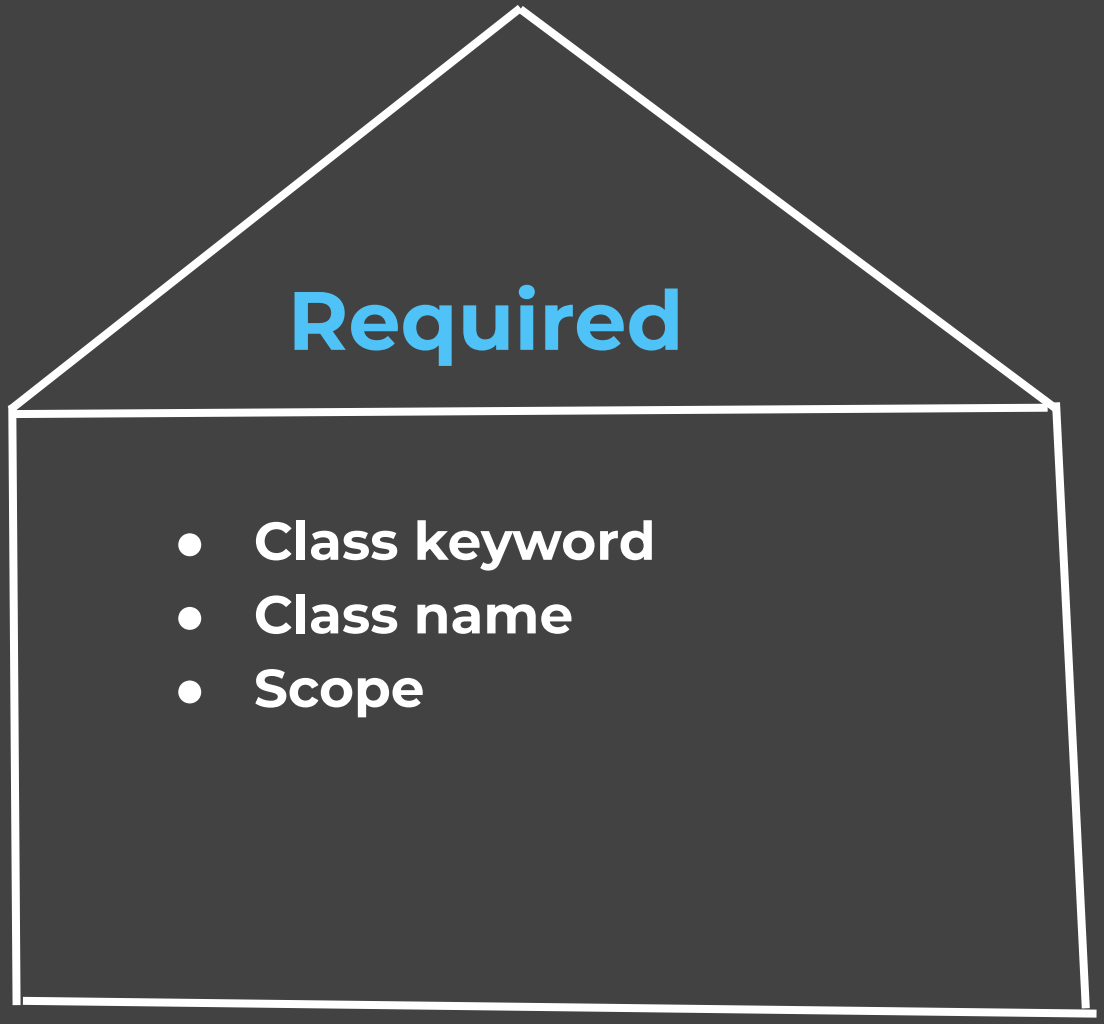
Classes are like a Blueprint

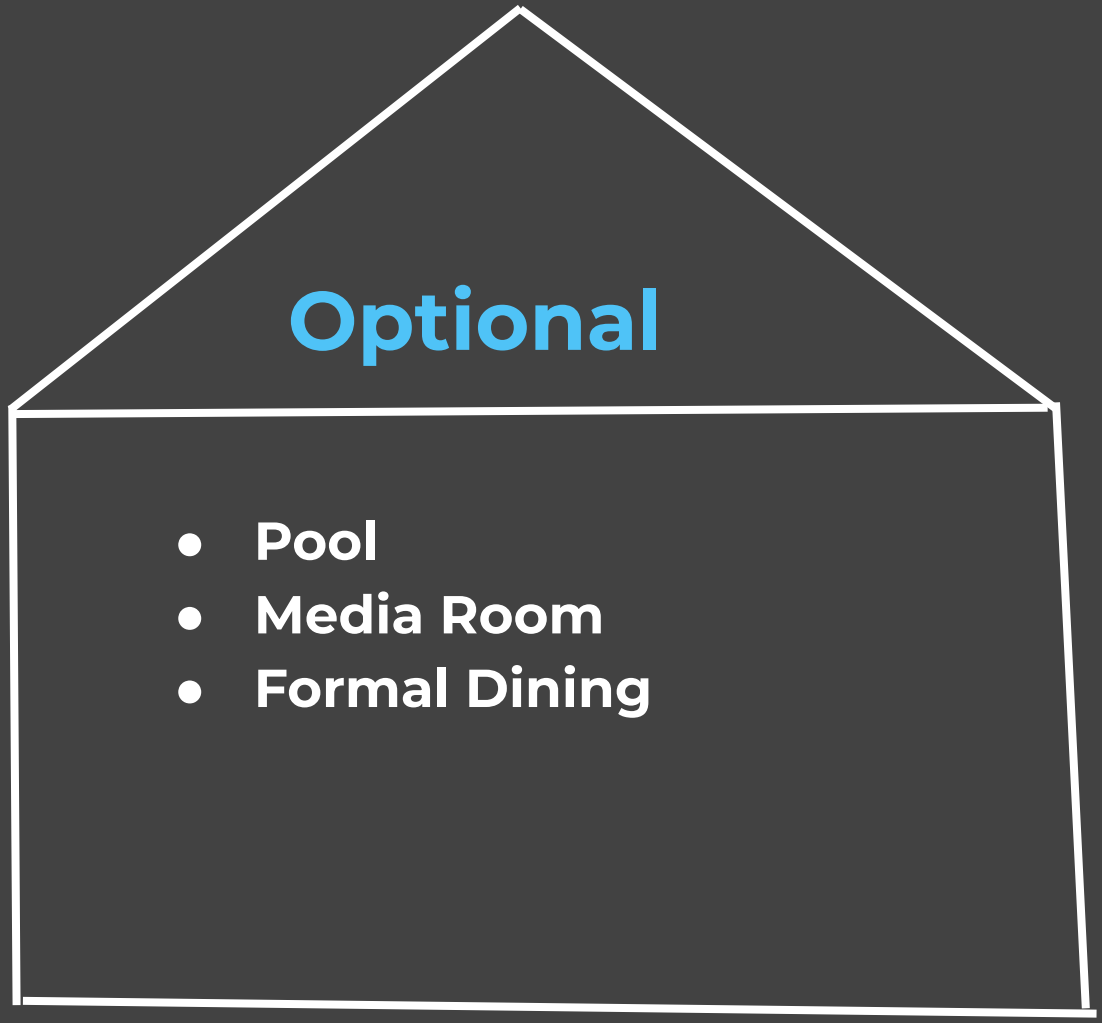


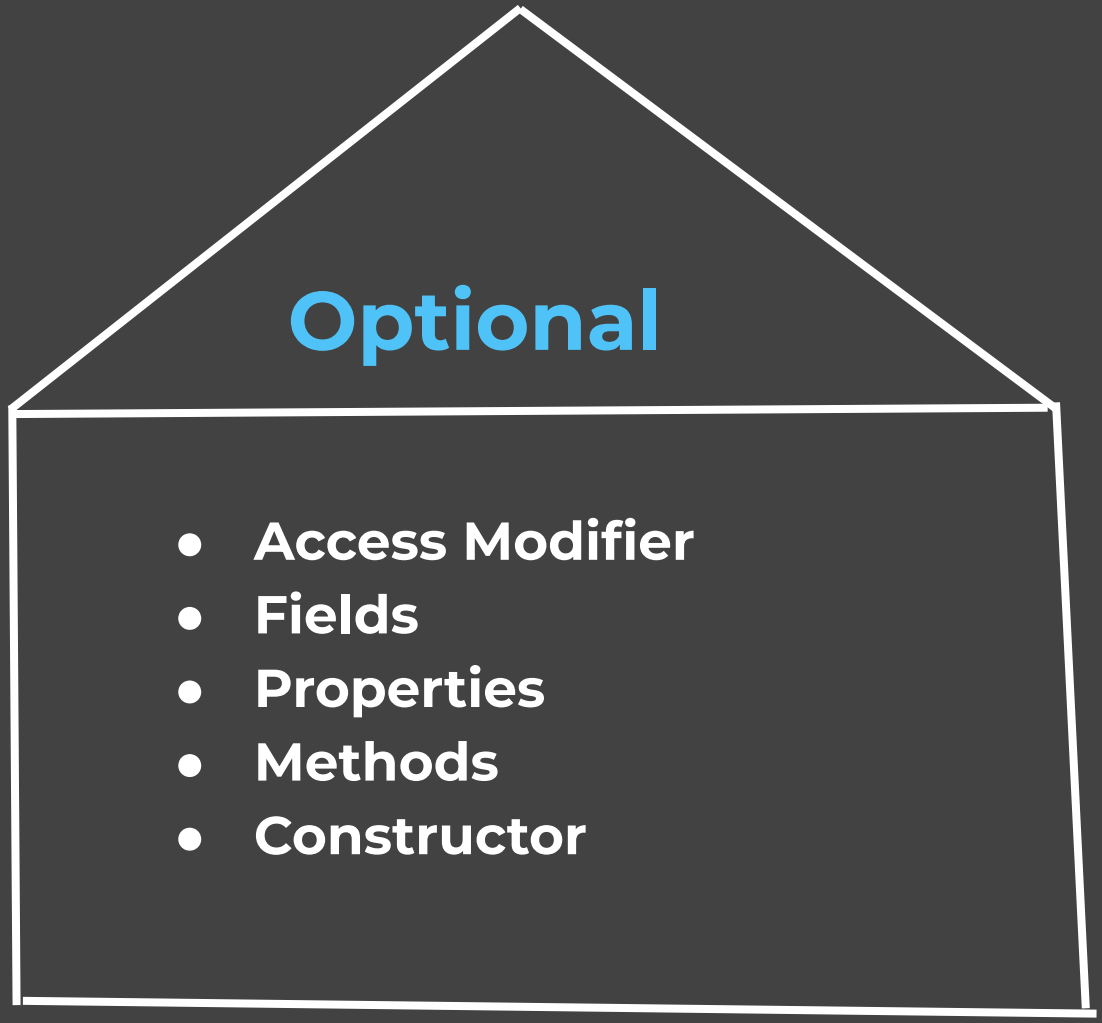

House
Bedrooms
Bathrooms
Kitchen
Pool


Class members
Fields
Properties
Methods
Constructors


Required
Roof
Plumbing
Walls
etc


Required
Class keyword
Class name
Scope


Optional
Pool
Media Room
Formal Dining


Optional
Access Modifier
Fields
Properties
Methods
Constructor

Example of a Class

In this class we have a:

- Constructor
- Field
- Property
- Method

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

Field

- Like a variable, but belongs to the class
- It is declared directly inside a class
- Can access it through methods

It has:

1. Access modifier
2. Type
3. Variable name

```
public class Dog
{

    private int _numberOfLegs;  // Field

}
```

Property

Allows us to “get” and “set” information

Get → Read

Set → Write

```
public class Dog
{

    public string Name { get; set; } // Property

}
```

Full Methods

Lots of lines!

```
private string _dogName;

public string GetDogName()

{

    return _dogName;

}

public string SetDogName(string dogName)

{

    _dogName = dogName;

}
```

Full Property

```
private string _dogName;

public string DogName()

{

    get {

            return _dogName;

         }

    set

        {

            _dogName = value;

        }

}
```

Properties

```
private string _dogName;

public string DogName()

{

  get { return _dogName };

  set {_dogName = value};

}
```

Properties

```
private string _dogName = “User Name”;

public string DogName()

{

  get { return _dogName };

  set {_dogName = value };

}
```

Properties

Auto-Implemented Properties

-Use Pascal Casing

```
public string Name { get; set; }
```

Properties

Auto-Implemented Properties

-How to set a default value

```
public string Name { get; set; } = “User Name”
```

Shortcut

- Property : prop + tab + tab

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

Constructor

When we instantiate a class, its constructor is called.

- It is a special member method
- No return type
- Has same name as the class

```
public class Dog
{
    public Dog()  // Constructor
    {
    }


}
```

NOTE:

- Macs will automatically create a default constructor upon creation of a class
- Windows will not automatically make one

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

Shortcut

Constructor: Ctor + Tab + Tab

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

You NEED a constructor to “build”/make an object

If you don’t make one, it will default to using the default constructor.

There a Two Types of Constructors

1. **Default**
2. **Custom**

Default Constructor

- No parameters

```
public class Dog
{
    public Dog()  // Constructor
    {
    }


}
```

Custom Constructor

- You define it yourself
- Has parameters

Benefit over Default Constructor:

-Allows you to set values upon creation of an object

```
public class Dog
{
    public Dog(string name, string breed)
    {
        Name = name;
        Breed = breed;
    }

    public string Name { get; set; } // Property

    public string Breed { get; set; } // Property


}
```

Optional: Access Modifier

If you do not specify, classes will default to internal, and methods will default to private

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

Optional: Constructor

If we do not write out a constructor, it will use the default constructor!

```
public class Dog
{
    //NO constructor written out

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

Optional: Constructor

- If we do not write out a constructor, it will use the default constructor!

```
public class Dog
{
    public Dog()  // Constructor
    {
    }


}
```

Constructor Overloading

Can use both default and custom – however, you must explicitly write out both

(if you just write out the custom, it won't let you use the default)

```
public class Dog
{
    public Dog()  // Default
    {
    }

    public Dog(string name, string breed)    //Custom
    {
        Name = name;
        Breed = breed;
    }


}
```

Constructor

Just like methods – How does it know which constructor to invoke?

```
Dog dog1 = new Dog() // Default

Dog dog2 = new Dog(“Frassy”, “Lab”);    //Custom
```

Like Method Overloading!

It knows which constructor to invoke based on the **parameters**

```
public class Dog
{
    public Dog()  // Default
    {
    }

    public Dog(string name, string breed)    //Custom
    {
       Name = name;
       Breed = breed;
    }


  }
```

Classes are Templates!

This is not an actual dog

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

How do we create a dog?

How to: Create a dog

Dog 1:

Dog 2:

```
Dog dog1 = new Dog();    //this is an instance of a dog

Dog dog2 = new Dog();    //this is our second dog


                         //We have TWO objects or TWO dogs
```

How to: Create a dog

Dog 1:

Dog 2:

```
Dog dog1 = new Dog();    //this is an instance of a dog

Dog dog2 = new Dog();    //this is our second dog


                                //We have TWO objects or TWO dogs
```

```
//Instance == Object
```

Our “dog” has access to:

- Everything in this class

```
public class Dog
{
    public Dog()  // Constructor
    {
    }

    private int _numberOfLegs;  // Field

    public string Name { get; set; } // Property

    public void Speak()  // Method in the class
     {

      Console.WriteLine(Name);

     }

  }
```

Just like lists!

-Just like we instantiate a list, we can instantiate a class

```
List<int> myList = new List<int>();

Dog dog1 = new Dog();
```

Initialize

Just like we initialize a list, we can initialize a class

```
List<int> myList = new List<int>() { 1, 2, 3};

Dog dog1 = new Dog() {Name = “Sassy”, Breed = “Lab” };
```

Ways to initialize members members a class

1. **Dot notation**
2. **Object initializer syntax**
3. **Custom constructor**

1. Dot Notation

```
Dog dog1 = new Dog();    // instance

dog1.Name = “Sassy”;      //Setting the property

dog1.Breed = “Lab”;       //Setting the property


Dog dog2 = new Dog();    // instance

dog2.Name = “Frassy”;      //Setting the property

dog1.Breed = “Lab”;       //Setting the property
```

2. Object Initializer Syntax

Object initializer syntax is a way to initialize an object and its properties at the time of creation without explicitly invoking a constructor for each property.

```
Dog dog1 = new Dog(){Name = “Sassy”, Breed = “Lab”};
```

```
Dog dog2 = new Dog(){Name = “Frassy”, Breed = “Lab”};
```

2. Object Initializer Syntax

You can spread it out so it's more readable

```
Dog dog1 = new Dog()

{

   Name = “Sassy”,

   Breed = “Lab”

};


Dog dog2 = new Dog()

{

   Name = “Frassy”,

   Breed = “Lab”

};
```

3. Custom Constructor

**Reminder what a custom constructor looks like

```
public class Dog
{
    public Dog(string name, string breed)
    {
        Name = name;
        Breed = breed;
    }

    public string Name { get; set; } // Property

    public string Breed { get; set; } // Property


}
```

3. Custom Constructor

- How to invoke a custom constructor
- You can immediately set values when you create an instance

```
Dog dog1 = new Dog(“Sassy”, “Lab”);   // instance
```

Instance Method

instanceName.MethodName();

```
Dog dog1 = new Dog();    //this is an instance of a dog

dog1.Speak();            //call the method for dog1
```

Classes Demo

Pinning VS Community 2022 to Taskbar

Creating a New Project in VS Community 2022