Understanding Classes and Objects

Classes in C#

As you've already learned, C# depends heavily on types. It's known as a *strongly-typed*language, because it enforces type definitions and constraints. So, for instance, if a method has a parameter of type `string` , you can't just pass it a variable (or literal) defined as an `int` . C# will generate a compile error. The types you work with, both the built-in ones and your own custom types that you'll create, are defined (usually) by *classes*.

You can think of a class as a definition or design of an *object*. **In C#, an instance of a class is called an object.** The class definition specifies what kinds of state the object will contain, and how it will expose that state to other objects. In this way, you can think of a class as simply a data structure. But classes are often much more than just dumb containers of data. They also include methods that define behavior, and the combination of data or state with behavior that can operate on that state is a powerful concept in object-oriented programming.

In your C# programs, just about every file that ends in ".cs" will be a class definition. You may also have some *interfaces*, *structs*, *enums*, or other elements, but classes usually make up the vast majority of C# applications. You've seen quite a few versions of the standard console application class, typically called "Program.cs", already. Its definition looks like this:

```
public class Program
{
}
```

Defining classes in C# is very straightforward. You'll usually want the class to be accessible anywhere, so you'll declare it as `public` . You'll learn more about how these modifiers work in the next lesson. Next, the `class` keyword is specified, followed by a valid C# name for the class. That's all that's required to define a simple class.

To instantiate, or create an instance of, a class, you use the `new` keyword. Within a console application's `static` main method, you don't have access the `Program` 's non-static members (if any). However, you can create an instance of `Program` from within the `Main` method, and then work with that instance and its members. For example:

```
public class Program
{
     int versionNumber = 123; // defines a field

     public static void Main()
     {
         // the following line will not compile - comment out to fix
         Console.WriteLine($"Current version: {versionNumber}");
```

```
        // instead, you must create an instance of Program
        var myProgram = new Program();
        Console.WriteLine($"Current version: {myProgram.versionNumber}");
    }
}
```

```
using System;

public class Program
{
    int versionNumber = 123; // defines a field

    public static void Main()
    {
        // the following line will not compile - comment out to fix
        Console.WriteLine($"Current version: {versionNumber}");

        // instead, you must create an instance of Program
        var myProgram = new Program();
        Console.WriteLine($"Current version: {myProgram.versionNumber}");
    }
}
```

By default, all classes *inherit* from the `System.Object` type in .NET, which provides some behavior that all classes can use or modify. For example, `System.Object` defines a method `ToString`, which means you can call this method on any object in all of .NET and expect a string result representing the object in some fashion. Working from the example above, you can display the default `ToString` output of your `Program` class with:

```
var myProgram = new Program();
Console.WriteLine(myProgram.ToString());
// or
Console.WriteLine(myProgram); // WriteLine will automatically call ToString for you
internally
```

```
using System;

public class Program
{
    int versionNumber = 123; // defines a field

    public static void Main()
    {
       var myProgram = new Program();
       Console.WriteLine(myProgram.ToString());
       // or
       // WriteLine will automatically call ToString for you internally
       Console.WriteLine(myProgram);
    }
}
```

In this case, you'll get output like the following (twice):

```
Program
```

The default implementation of `ToString` is to simply display the full name of the type, which is the namespace (in this case, `ConsoleApplication`) and class name (`Program`). However, you can *override* this default behavior in your classes if you wish to change the behavior to something more useful. You'll learn more on this later in this lesson.

Properties and Fields

Classes have *members*, which consist of methods, properties, and fields. You learned about methods in the previous lesson. You saw an example of a field in the previous section, in which a `versionNumber` integer was added to the `Program` class. *Fields* are types that are attached to a class. They track the state of the class, and separate instances of the same class will each track the data of their fields independently. *Properties* provide a way for other objects to access state from an object in a controlled manner. Unlike fields, which are essentially just variables, properties are methods and can add additional behavior around manipulating the state of an object.

For example, suppose you have a class representing a speedometer, which can display values from 0 to 120. You implement the class using a field to represent the current speed:

```
public class Speedometer
{
    public int CurrentSpeed; // uninitialized ints start out with value of 0
}
```

This class, in turn, controls the dashboard of the vehicle, which only understands inputs between 0 and 120. Other values for `CurrentSpeed` , such as negative numbers, may result in unexpected behavior. As written, there's no way to ensure that `CurrentSpeed` is never set to a value that is out of range, which means the code that interfaces with the dashboard controls will need to be full of checks to ensure the values are appropriate. Wouldn't it be better to ensure no bad values were possible, by limiting how CurrentSpeed is set? One way to do this is to use methods:

```
public class Speedometer
{
    private int _currentSpeed;
    public int GetCurrentSpeed()
    {
        return _currentSpeed;
    }
    public void SetCurrentSpeed(int newSpeed)
    {
        if (newSpeed < 0) return;
        if (newSpeed > 120) return;
```

```
            _currentSpeed = newSpeed;
        }
    }
```

Using methods to make updates to the state of an object is often preferred, since it allows you to add any necessary business logic surrounding what kinds of updates are allowed. Properties are a C# feature that allows you to define special `get` and `set` methods for any state you want your objects to expose to other objects (or even for their own use). Modifying the above example to use a property instead, you would rewrite it as follows:

```
public class Speedometer
{
    private int _currentSpeed;
    public int CurrentSpeed
    {
        get
        {
            return _currentSpeed;
        }
        set
        {
            if (value < 0) return;
            if (value > 120) return;

            // value is a keyword used in setters representing the new value
            _currentSpeed = value;
        }
    }
}
```

> **Tip** {.tip .java}
> Java uses conventions for its accessor and mutator methods; C# uses `get` and `set` to achieve this purpose, with properties as a dedicated language construct.

You should rarely expose fields from your classes, and instead use properties to control external access to your object's state. Even then, you should limit such access, so that your object's collaborators do not become too tightly coupled to your object's internal implementation. In cases where you don't need to perform any additional behavior, your property definitions can be very simple:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string TaxPayerId { get; set; }
}
```

Constructors and Property Initializers

Note that in the above example, `Person` 's properties all start out uninitialized. Thus, the `string` properties will be `null` , and the `DateOfBirth` property will initially be `DateTime.MinValue` , which probably isn't what's desired. You can specify default values for a class's members in its *constructor*. The constructor is run when an instance of the class is created, before any code can interact with the new instance. You should avoid putting complex logic into constructors, but they're a good place to ensure fields and properties are initialized. You can see an example of a `Person` constructor below:

```
public Person() // this is a method within the Person class
{
    FirstName = string.Empty;
    LastName = string.Empty;
    TaxPayerId = string.Empty;
}
```

Constructors can also take parameters. By default, all classes have a default constructor that takes no parameters. If you add a constructor, the default constructor will no longer be created, so you can control whether or not a class can be constructed without parameters. In this case, pretend you want to require a `DateOfBirth` when creating a `Person` . In that case, change the above `Person` constructor as follows:

```
public Person(DateTime dateOfBirth)
{
    DateOfBirth = dateOfBirth;
    FirstName = string.Empty;
    LastName = string.Empty;
    TaxPayerId = string.Empty;
}
```

Putting property initialization code into constructors can result in a lot of extra code, and for larger classes there can be a lot of separation between where the property is declared and where its value is initialized. That's why C# recently added support for *property initializers*. Property initializers can be used to set the default value of properties at the same time they're declared. The syntax is straightforward: after the automatic property declaration, add an `= initialValue;` where *initialValue* is what you want the property to be. For example, setting the string properties to default to empty strings (instead of null), while still requiring `DateOfBirth` in the constructor would be done as follows:

```
public class Person
{
    public Person(DateTime dateOfBirth)
    {
        DateOfBirth = dateOfBirth;
    }

    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
    public DateTime DateOfBirth { get; set; }
    public string TaxPayerId { get; set; } = string.Empty;
}
```

Composition

Fields and properties provide a way for you to *compose* your objects from other objects. This gives you a powerful way to add and share properties and behavior between objects. For instance, let's say you needed to mail things to people and companies as part of your program. You could put all of the necessary properties on both the `Person` and `Company` classes:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string StreetAddress { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}
public class Company
{
    public string Name { get; set; }
    public string StreetAddress { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}
```

However, a better approach would be to pull out the properties that logically can be thought of as an *address*, and compose the other classes using this new type:

```
public class Address
{
    public string StreetAddress { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address ShippingAddress { get; set; }
}
public class Company
{
    public string Name { get; set; }
    public Address ShippingAddress { get; set; }
}
```

Another approach to share this state between two different classes would be to use *inheritance*, which you'll learn about in a moment. However, it's generally better to **favor composition over inheritance** in your C# applications, as it results in a more flexible design.

Inheritance

In object-oriented languages like C#, classes can *inherit* from other classes. You've already learned that all classes in C# inherit from `System.Object`, but they don't necessarily do so directly. Just as you inherit from your grandparents, but through your parents, so too can your classes inherit from other classes before ultimately inheriting from `System.Object`. Inheritance provides a form of reuse, because common state and behavior characteristics of objects can be defined in *base* or *parent* classes and then used by *child* classes, either directly or with further refinement.

In some languages, classes can inherit from multiple base classes, pulling in behavior from multiple parents. However, C# supports *single inheritance*, meaning that a class can only inherit from one base class. This simplifies many aspects of the language, but does mean that you need to be careful with how you use inheritance in your application design, because if you choose to have a class inherit from one class, you can't later choose to have it *also* inherit from another class. This is another reason to favor composition over inheritance, since there's no limit to how many other classes you can reference in your classes via fields or properties.

A simple example to demonstrate inheritance is one that uses geometric shapes. You can define a class, `Shape`, that includes methods for calculating values like Perimeter. Then, you can inherit from `Shape` with various specific kinds of shapes, implementing the methods as you do so.

```
public class Shape
{
    public virtual int Perimeter()
    {
        return 0;
    }
}
public class Rectangle : Shape
{
    public int Height { get; set; }
    public int Width { get; set; }

    public override int Perimeter()
    {
        return (Height + Width) * 2;
    }
}
public class Triangle : Shape
{
    public int Side1 { get; set; }
    public int Side2 { get; set; }
    public int Side3 { get; set; }
    public override int Perimeter()
    {
        return Side1 + Side2 + Side3;
    }
}
```

You'll notice two new keywords in the example above: `virtual` and `override`, as well as some new syntax in the class definition lines. You specify the class your class inherits from by specifying it after a colon (`:`) following the class's name. You can actually add `: System.Object` to any class that doesn't have another base class without changing its behavior - this is implied within .NET. Classes that inherit from other classes expose the base class's methods to their collaborators. Any code that refers to a `Shape` will be able to call the `Perimeter` method. By declaring that method as `virtual`, child classes can modify the behavior of the method; by default methods cannot be changed by child class implementations. When changing base class behavior, the child class defines the method with the same return type, name, and signature, as well as the `override` keyword. Requiring the use of the `override` keyword ensures that developers do not accidentally override base class behavior.

> **Tip** {.tip .java}
> The C# `:` operator is equivalent to `extends` and `implements` in Java. Like Java, C# does not support multiple inheritance.

> **Tip** {.tip .java}
> Unlike Java, C# methods are not virtual by default - you must explicitly mark as virtual those members that can be overridden by child types.

Note that you can override the `ToString` method on any class where it will make sense to display string data related to an instance of the class. Returning the first example in this lesson, you could display the version of the program along with its name by overriding `ToString`, like so:

```
public class Program
{
    int versionNumber = 123; // defines a field

    public override string ToString()
    {
        return $"MyProgramName - Version {versionNumber}";
    }
}
```

```
using System;

public class Program
{
    int versionNumber = 123; // defines a field

    public static void Main()
    {
    var myProgram = new Program();
```

```
        Console.WriteLine(myProgram.ToString());
        // or
        // WriteLine will automatically call ToString for you internally
        Console.WriteLine(myProgram);
        }

        public override string ToString()
        {
            return $"MyProgramName - Version {versionNumber}";
        }
    }
```

Abstract types

Sometimes, you may define a class that should never actually be instantiated as an object - it should only be used as a base class for others. These are referred to as *abstract* classes. In the example above, it doesn't really make a lot of sense to create an instance of `Shape`, which has no sides and which has a `Perimeter` method that always returns 0. It would probably make more sense to define the `Shape` class as `abstract`. You can also define methods as `abstract`, so that their implementation is required in child classes, but unnecessary in the base class. An abstract version of the `Shape` class would be:

```
public abstract class Shape
{
    public abstract int Perimeter();
}
```

Note that `abstract` methods do not require the `virtual` keyword, since they *must* be overridden in child class implementations (the code will not compile otherwise).

Polymorphism

Polymorphism means, "occurring in several different forms". In programming, polymorphism refers to the ability to have code that can work with objects of different forms as if they were the same. Polymorphism and inheritance are related, because you should be able to substitute child types for their base types anywhere you need to work with them (known as the Liskov Substitution Principle). This means that you can write methods that accept a base type as a parameter, and they can work with that base type's methods and properties, regardless of what *actual* child implementation of that base class is passed as an argument. For instance, to display the perimeter of a shape, you could write a method like this one:

```
public void DisplayShape(Shape shape)
{
    Console.WriteLine($"Shape Perimeter: {shape.Perimeter()}");
}
```

The ability to have the `DisplayShape` method work correctly with many different `Shape` implementations is an example of polymorphism. This allows the calling code and the specific implementation of `Shape` to evolve independently from one another, resulting in a more maintainable program.

To see an example of this in action, consider the following code block:

```
var rectangle = new Rectangle();
rectangle.Height = 5;
rectangle.Width = 6;
DisplayShape(rectangle);

var triangle = new Triangle();
triangle.Side1 = 3;
triangle.Side2 = 4;
triangle.Side3 = 5;
DisplayShape(triangle);
```

Both `Rectangle` and `Triangle` object instances can be passed as arguments to `DisplayShape` , since it expects a `Shape` instance, and both of these types inherit from `Shape` .