Object Oriented Programming

Table Of Contents

Preamble

This set of slides is one I would recommend looking back on after this week. It has a lot of concepts within it which may not be immediately applicable to the exercises you are going to utilise but necessary further on.

Classes

There is a clear distinction between a primitive type and a reference type but how is the distinction made?

Reference Types are Classes

We have already used classes within our programs since the start of semester. However now we are able to define our own classes.

Most programming languages have some mechanism of structuring data for reuse.

Classes - But What are they?

> "A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviours. When the program is run, each object can act alone or interact with other objects to accomplish the program’s purpose."

Sometimes it is simply conveyed as a blueprint/template/concept of an **object**.

C# as an OOP language, primarily focuses on this way to structure data and code.

Classes

You have used classes in some form already and also with some basic usage of methods.

What we have missed is instantiating something that is more explicitly or not simply derived to be a class (like an array).

You have used classes like Console, String and Array.

Classes - Objects

The **type** of an object variable is its **class**.

```
Point p;
```

The class is `Point` which we will also call its **type** as well.

We can instantiate a class by using the `new` keyword and like so.

- `Point topleft = new Point(-1, -1);`
- `Point right = new Point(1, 0);`
- `Point home = new Point(-3388797, 15119390);`

Classes - `new` keyword

The `new` keyword does some amount of heavy lifting here. Whenever we use `new`, we are being explicit with allocating memory to hold this data.

- Since we have a point that holds at least two ints, we need to store that somewhere.

You will observe it is used in conjunction with the class name itself. Which can seem confusing but this relates to the **constructor** which outlines how it is built.

Classes - Can I make my own?

Yes!

However let's start off with a basic class definition.

```
class Cupcake {
  bool delicious = false;
  string name = "Chocolate Cupcake";
}
```

We can instantiate this class with the following line of code.

```
Cupcake c = new Cupcake();
```

Classes - Constructors

In the previous example, what was happening is that we have fields already initialised and usable, we are then able to use a **default constructor**.

Classes - Constructors

Extending our Cupcake class we can write our own constructor.

```
public class Cupcake {
    bool delicious = false;
    string name = "Chocolate Cupcake";

    public Cupcake() { /* NO OP */ }
}
```

While not much as changed, we now have some control over how it is constructed and we can start parameterising this **constructor**.

Classes - Constructors

Within C#, every class has a constructor, even if it is not explicitly defined.

However, extend or own constructor, we can initialise the fields within the constructor itself.

```
public class Cupcake {
    bool delicious;
    string name;

    public Cupcake() {

        delicious = true;
        name = "Chocolate Cupcake";
    }
}
```

Classes - Constructors

Since a constructor is just a special kind of static method. We are able to parameterise it.

```
public class Cupcake {
    bool delicious;
    string name;

    public Cupcake(bool isTasty, string cakeName) {
        delicious = isTasy;
        name = cakeName;
    }
}

Cupcake cake = new Cupcake(true, "Vanilla");
```

Classes - Encapsulation

Before we jump into methods and properties, we are going to look into encapsulation. Encapsulation within C# and other languages are a mechanism to restrict access to particular fields (as well as methods and properties).

These include the following keywords:

- `public` - No access restriction
- `private` - Access limited to the containing type
- `protected` - Access limited to the containing class or types derived from the containing class
- `internal` - Access limited to the current assembly (Useful for libraries).

Classes - Encapsulation

Commonly `public` and `private` are utilised, usually there is a clear idea where we want fields or methods to be accessible or not.

```
public class Cupcake {
    private bool delicious;
    private string name;
}
```

The above fields are restricted to the class scope although we will be able to show we can mediate how data is updated and retrieved from this type later on.

Classes - Encapsulation (without access modifiers)

In the abscence of an access modifier keyword, there are some particular rules.

It is a little more nuanced. Fields and Methods are usually considered private by default however classes are just considered internal.

- Classes will be accessible by other others and types within the same project/library but not outside of it by default (unelss specified as public).
- Fields, Methods and Properties are considered private by default.

However memorising these rules is mostly trivia, being explicit regarding access modifiiers usually makes it clear for other developers reading your code.

Classes - Encapsulation

It also means that, in the example prior, the fields have been non-accessible outside of the scope of the class. If we were to change this to `public`, we can access them outside.

```
public class Cupcake {
    public bool delicious; // Can now be access outside
    public string name;
    public Cupcake(bool isTasty, string cakeName) {
        delicious = isTasy;
        name = cakeName;
    }
}
Cupcake cake = new Cupcake(true, "Vanilla");
string cupcakeName = cake.name;
bool cupcakeDelicious = cake.delicious;
```

Classes - Methods

We're going to now visit the idea of methods, we have used `static` methods before but we will be using `instance` methods which operate on an object's fields.

Similar to static methods, we can construct a method that can use the data.

```
public class Cupcake {
    private bool delicious;
    private string name;
    // Rest snipped
    public string GetName()
    {
        return name;
    }
}

Cupcake cake = new Cupcake(true, "Vanilla");
string cupcakeName = cake.GetName();
```

Classes - Methods

Instance methods are a construct in which, we are associating code with an instance. Where static methods are are associated with a class/type, instance methods are associated with an object.

Methods are incredibly useful and common mechanism to group with methods with data. Typically we want to have code that associated with an instance/object.

Classes - Methods

As for some common use-cases for methods in general.

- Getters and Setters.
- Queries that may need to be computed or searched over.
- Actions that may need to be perform on an object or using an object.

... There is a lot of different patterns that can arise from this construct, simply put, if it can be represented as a static method/function, it can be also represent as an instance method as well.

Classes - `this` keyword and mixing methods

We'll expand on the this keyword and how it can help with eliminating ambiguity but also used for passing an object reference within an instance context.

The this keyword allows the programmer to refer to the object while within an instance method context. We cannot use the keyword within a static context.

Classes - `this` keyword

Let's use the following code snippet.

```
public class Postcard {
    string sender;
    string receiver;
    string address;
    string contents;

    public Postcard(string sender, string receiver, string address, string contents) {
        //Blasts! Foiled by ambiguity!
    }
}
```

Classes - `this` keyword

Now, a solution could be to simply rename the parameters to avoid the ambiguity.

```
// Rest snipped
public Postcard(string s, string r, string a, string c) {
    sender = s;
    receiver = r;
    address = a;
    contents = c;
}
```

However, this is undesirable, it impacts readability by exchanging it for cryptic letters.

Classes - `this` keyword

We can actually allieviate the ambiguity by using the `this` keyword.

```
public class Postcard {
    string sender;
    string receiver;
    string address;
    string contents;

    public Postcard(string sender, string receiver, string address, string contents) {
        this.sender = sender;
        this.receiver = receiver;
        this.address = address;
        this.contents = contents;
    }
}
```

So, what is the `this` keyword?

Classes - Mixing Static and Instance Methods

We covered instance methods in the previous lecture but now let’s expand on them and discuss about static and instance contexts.

We will be revisiting the this keyword again in this section to help understand how it is applied.

Within the context of an instance method, it refers to the current calling object. It cannot be used within a static method as it is unable to refer to the calling object.

Classes - Mixing Static and Instance Methods

Let's use the following snippet

```
public class Postcard {
    public string sender;
    public string receiver;
    // <...snip...>

    public void SetSender(string sender) {
        this.sender = sender;
    }
}
```

We can see how it can be used within a **setter** method. However, what if we have a static method instead and we want to avoid duplication?

Classes - Mixing Static and Instance Methods

Let's use the following snippet, we can observe we don't get the benefit from using this method directly with the object itself.

```
public class Postcard {
    public string sender;
    public string receiver;
    // <...snip...>

    public static void AssignSender(Postcard p, string sender) {
        p.sender = sender;
    }
}
```

Classes - Mixing Static and Instance Methods

However, we can add a method that can use this as well, not requiring us to reimplement it.

```
public class Postcard {
    public string sender;
    public string receiver;
    // <...snip...>
    public void SetSender(string sender)
    {
        Postcard.SetSender(this, sender);
    }

    public static void SetSender(Postcard p, string sender)
    {
        p.sender = sender;
    }
}
```

Classes - Static and Instance Methods

When and why would we use one over the other?

While it isn't difficult, there isn't a perfect "catch-all" rule but there are good guidelines and ideas to consider when designing your classes.

A reasonable guide would be to do the following.

- Code that is acting upon fields should be an instance method. (Most types)
- Code that is part of creating an object (constructor or static method) or in part utility of the type is best assigned as a static method. (Case Study: File type in C#)

Classes - Properties

While we have examined fields recently, we have not had much of a chance to discuss properties in C#. This construct is a bit of an in-between of a field and a method. Per-convention, the properties end up being a little more preferred in C# over just regular fields.

They end up being a bit of an inbetween fields and methods with some perculiar rules.

```
public class Person
{
  public int Age { get; set; }
}
```

Classes - Properties

There are a few components to properties that have to be understood as there is different behaviour depending how it is constructed. However two common properties that are used.

- Automatically Implemented Properties

These are properties in which the storage is allocated and known.

- Field Backed Properties

The other construction uses a field as part of storage.

Classes - Properties

Automatically implemented properties, these are where `get` and `set` keywords are used and enclosed.

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

Does encapsulation apply here?

Yes! We are able to specify access modifiers against `get` and `set`.

Classes - Properties

Field-Backed properties is where the class has a field in which either the `get` and `set`. To demonstrate, we are able to specify a field and use with `get` and `set` to retrieve or set the `value`.

```
private string firstName = String.Empty;
private string lastName = String.Empty;

public string FullName
    {
        get => firstName + " " + lastName,
        set {
            string[] parts = value.Split(" ");
            firstName = parts[0];
            lastName = parts[1];
        }
    }
```

Classes - Method Overloading

What is overloading? In regards to C# we are able to use the same method name but with different method signature. We are able to define a method such as add and have a version that accepts two integers and another version that accepts three integers.

```
int Add(int a, int b);

int Add(int a, int b, int c);
```

Classes - Method Overloading

You may have observed that when using Console.WriteLine that it was able to convert your data elegantly. You didn't have to convert your datatypes to strings unless it was a custom object. This is because WriteLine and Write leverage overloading here.

```
int Add(int a, int b);

int Add(int a, int b, int c);

float Add(float a, float b);
```

Classes - Method Overloading

A key observation here is how the parameters are required to be different, either different number of parameters or different types to clearly outline what particular method we want to use.

So, referring to those `Add` methods from before, given the examples below, what method would be selected for each call?

- `Add(1, 2)`
- `Add(1.5, 2.0)`
- `Add((int) 1.5, 2)`
- `Add(1, 2, 3)`

Namespaces

In a classical usage of namespaces in C#, the scoping idea is similar to that of classes and methods. We use the braces to indicate what types are grouped under a namespace.

```
namespace MusicLibrary {
    // Everything encompassed inside the braces is under this name
}
```

Namespaces allow for the grouping of types and other components underneath a name. This can be declared at either a file level or leverage scope as outlined before.

```
namespace MusicLibrary; // File level
```

Namespaces

The `using` keyword is used to include a namespace within the context of a file. This is normally to make it clear what types or components you want access to without needing to provide their full-qualified name.

```
using MusicLibrary;

//...

    static void Main(string[] args)
    {
        Album a = new Album("Dive", "Tycho", "2016");
        // Album is part of MusicLibrary
        // We would need to use `MusicLibrary.Album` if we didn't use
        // `using MusicLibrary;`
    }
```

References and Primitive Types

Arrays and Objects are reference types. As outlined in Index Calcuation, Arrays are assigned to a memory address, Objects are also assigned to a similar value or modelled where each field is an offset from the base address.

References and Primitive Types

As a refresher, some of the common C# primitive types are.

- int - Integer datatype, will represent numbers like 1, 2, -3, 50.
- bool - Represents true or false
- float - Floating point value, represents values like 20.5, 1.22, 9.67
- double - Similar to float but with more precision.
- byte - Integer value between -128 to 127.

However, `Array` and instances/objects are **reference types**.

Time to make things!