POWERING POTENTIAL

Object-oriented programming

CONTENTS

- **Objectives**
  - To understand types and how to create objects
  - To understand reference type behaviour

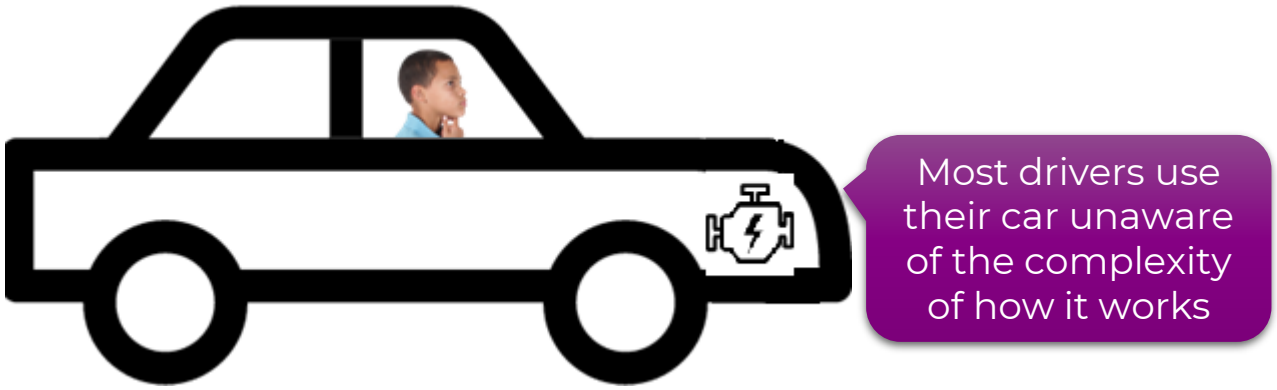
- **Contents**
  - OO Fundamentals – abstraction and encapsulation
  - Defining reference types
  - Creating objects (instances)

- **Hands-on labs**

OO Fundamental – Abstraction

- **Ability to represent a complex problem in simple terms**
  - Creation of a high-level definition
  - Factoring out common features of a category of objects

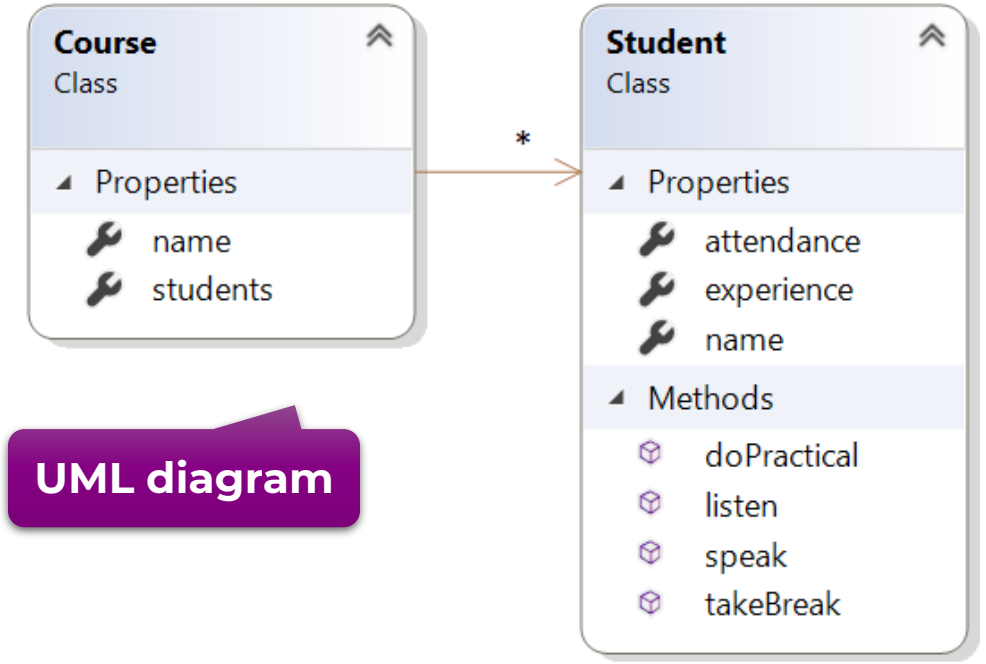
- **Stresses ideas, qualities & properties not particulars**
  - Emphasises what an object is or does, rather than how it works
  - Primary means of managing complexity in large app



An example of using abstraction

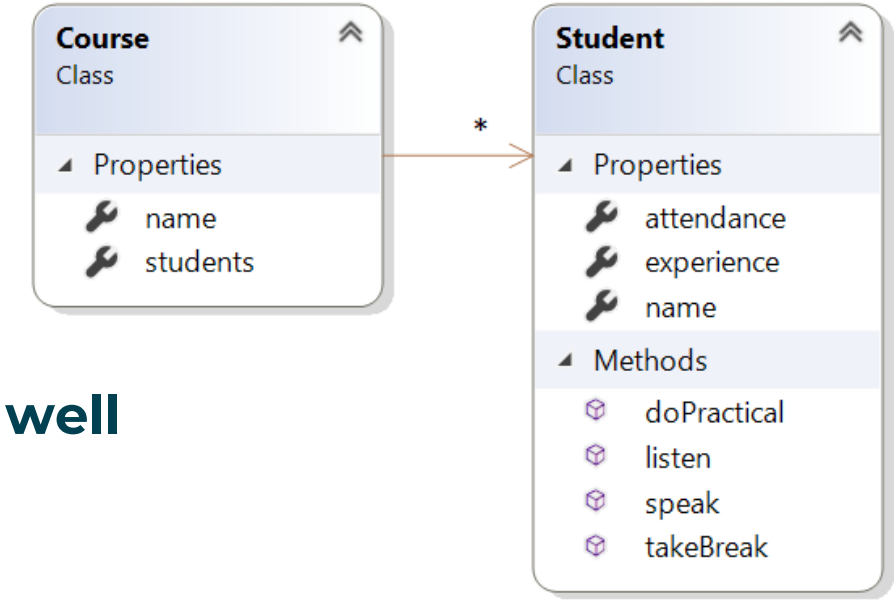
- **Students (instances of type Student) attending a Course**
  - Have '**attributes**' - name, experience, attendance record
  - Have '**behaviour**' - listen(), speak(), takeBreak(), doPractical()

  - Are part of '**relationships**'
    - A student 'attends a course,
    - a course 'has' many students



OO Fundamental – Encapsulation

- **Hiding object's implementation, making it self-sufficient**

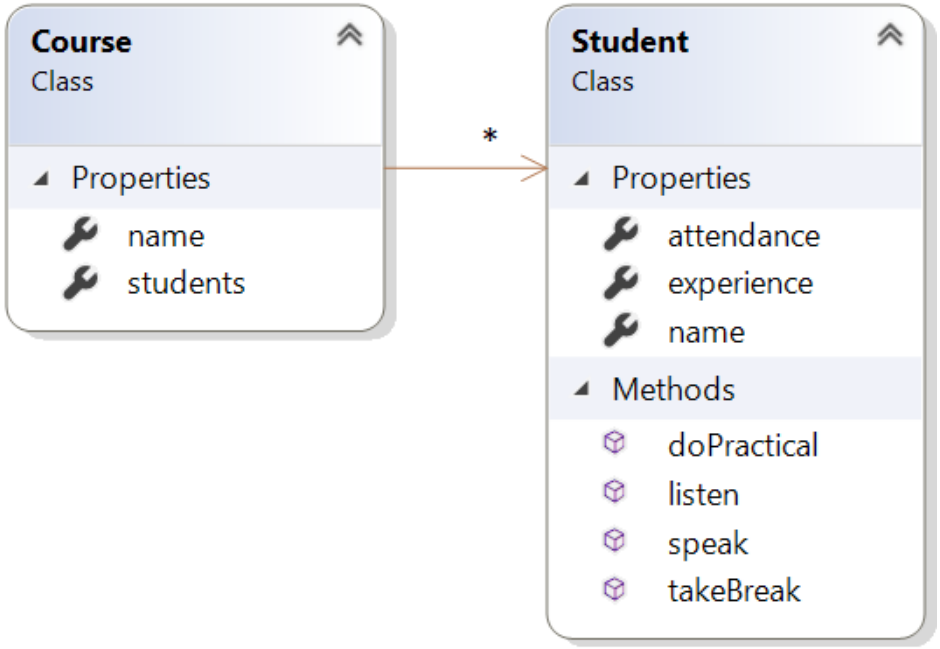


- **Process of enclosing code needed to do one thing well**
  - Put the data that code needs in a single object
  - Allows complexity to be built from simple objects
    - Internal complexities are hidden in the objects
      e.g. how a student performs a lab
    - Users of an object know its required inputs and expected outputs
    - Benefits reliability, maintainability and re-use

OO Fundamental – Objects communicate

- **Objects communicate via messaging (method calls)**
  - Messages allow (receiving) object to determine implementation

```
foreach(Student student in students)
{
   student.doPractical(2);
}
```



What is an OO data type?

A class definition is a blueprint for making *objects*

- **Fields** - Constituent data parts. **Hold state**
- **Methods** - Methods that define **behaviour**

Fields
state

Methods
Behaviour

```
public class Car
{
    string model;
    int speed;

    public void Start()
    {
        speed = 1;
    }

    public void Accelerate(int amount)
    {
        speed += amount;
    }
}
```

Public members are An object's interface

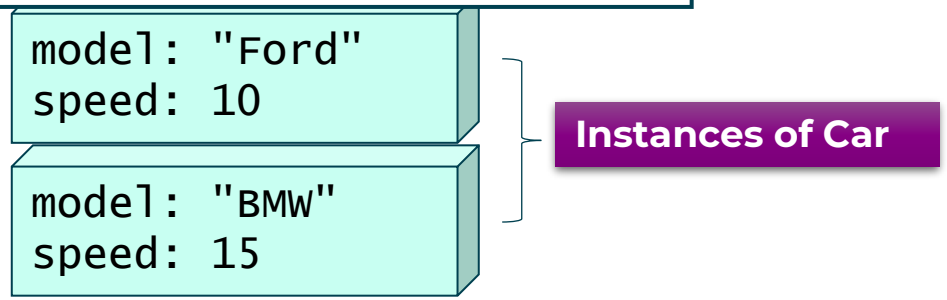
Creating Objects

- Objects are unique **instances** of a class with their own **state**.

Blueprint

```
public class Car
{
    string model;
    int speed;

    public void Start()
    {
        speed = 1;
    }

    public void Accelerate(int amount)
    {
        speed += amount;
    }
}
```

```
public static void Main(…) {

   Car car1 = new Car();
   Car car2 = new Car();

   car1.model = "Ford";
   car2.model = "BMW";

   car1.Accelerate(10);
   car2.Accelerate(15);
}
```



Getters and setters – Properties

- Do not expose state directly

```
public class Student
{
    private string name;
    private int age;
}
```

```
public static void Main(…) {

    Student student = new Student();

    student.name = "Bob"; ✖
    student.age = 25; ✖

}
```

Exposing state through property methods

```
public class Student
{
    řsîŵăţĵê îŋţ ăĝê
    řsîŵăţĵê ṣţĵsîŋĝ ŋăṇê

    řụčľîç ṣţĵsîŋĝ Ņăṇê

        ĝêţ   sêţĵụsŋ ŋăṇê
        ṣêţ   ŋăṇê   ŵăľụê


    řụčľîç îŋţ Aĝê

        ĝêţ   sêţĵụsŋ ăĝê
        ṣêţ

          îğ ŵăľụê
            ăĝê   ŵăľ�given
```

```
public static void Main(…) {

    Student stu = new Student();

    stu.name = "Bob"; ✖
    stu.age = 25; ✖

    stu.Name = "Bob"; ✓
    stu.Age = 25; ✓
}
```

Name and age are private

name is changed

Exposing state through auto-implemented properties

```
public class Student {

    řųčľîç îŋʃ Aĝê    ĝêʃ   ṣêʃ

    řųčľîç ṣʃsîŋĝ Ņắṇê    ĝêʃ   řsîŵắʃê ṣêʃ


    řųčľîç ŵộîđ Ṛêĝîṣʃês

        Ņắṇê     Bộč    ✓

}
```

```
public static void Main(…) {

    Student stu = new Student();

    int name = stu.Name; ✓

    stu.Name = "Bob"; ✘

    stu.Age = 25; ✓

    int age = stu.Age; ✓
}
```

Parts of .NET libraries do not consider state unless exposed using getters and setters

Object construction

- Let's consider two classes and the two instances created

```
public class Car
{
  private int speed;
  private string model;
}
```

```
Car myCar = new Car();
```

What model is this?
What is its speed?

```
public class Account
{
  private int id;
  private string owner;
}
```

```
Account myAccount = new Account();
```

What is the id of this account?
Who owns it?

We need a constructor

Constructor

The same name as the class. No return value. Not even void

```
public class Account
{
  private int id;
  private string owner;

  public Account (int id, string owner) {
      this.id = id;
      this.owner = owner;
  }
}
```

```
Account myAccount = new Account(123, "Bob");
```

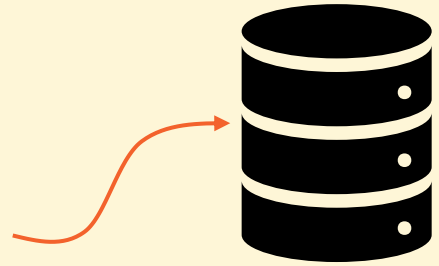
```
Account myAccount = new Account();
```

A default constructor does not exist. To create an instance of Account, you must provide the ID and the owner's name

Object construction - Overloading

- Overloading provides alternative ways for creating an instance

```
public class Account
{
  private int id;
  private string owner;

  public Account (int id, String owner)
  {
     this.id = id;
     this.owner = owner;
  }

  public Account (int id)
  {
     this.id = id;
     this.owner = GetOwnerById(id);
  }
}
```



```
Account myAccount = new Account(123, "Bob");
```

```
Account myAccount = new Account(123);
```

Constructor chaining example

```
public class Car
{
  private string model;
  private int numDoors;

1 public Car(string model, int numDoors)
  {
    this.model = model;
    this.numDoors = numDoors;
    // common code can sit here
  }

2 public Car(string model) : this(model, 4)
  {
  }
}
```

```
Car car1 = new Car("BMW",5);

Car car2 = new Car("BMW");

Car car3 = new Car();
```

Chaining, using special syntax

The null reference

```
public Car GetPoolCar() {
        Car car = null;

        // attempt to get a Car
           from a pool of cars

        return car;
}
```

car may not reference an object

```
public void HireCar() {
        Car myCar = GetPoolCar();

        if (myCar != null) {
                // Drive the car away
        } else {
                // No car available
        }
}
```

Can compare an object reference with null

What is a struct?

A struct is mainly used to package small number of other value types together

- Can be instantiated without using the **new** keyword.
- Can declare constructors with parameters but the **default one will always remain!**
- Can have fields, properties and methods just like classes
- Cannot inherit from another **struct** or class (**no OO**!)

- Microsoft has defined many structs such as **int** and **double** which you've used already!
  - It has many more, like *Rectangle, Point, Size, RectangleF, PointF, SizeF...*
  - **struct** types can be passed **by value** (by copy) from one method to another

- Live on the stack, not the heap, and are more efficient than reference types, but...
  - Do not define fields that are reference types (like array, List, or any other object)
  - Keep the overall size of the fields small

- There are few instances in the real world when you need to create a struct, but they do show up!

An example of a struct

```
řụčľîç ştʃsụçtʃ Bộụŋđ

    řụčľîç îŋtʃ Ŷ  Ÿ  Ẅ  Ȟ
    řụčľîç Bộụŋđ îŋtʃ ɣ  îŋtʃ ỳ  îŋtʃ x  îŋtʃ ḥ

        Ŷ   ɣ  Ÿ   ỳ  Ẅ   x  Ȟ   ḥ

    řụčľîç îŋtʃ Ṛîĝḥtʃ   ĝêtʃ   sêtʃụsŋ Ŷ   Ẅ
    řụčľîç îŋtʃ Bộţtʃộṇ   ĝêtʃ   sêtʃụsŋ Ÿ   Ȟ
    řụčľîç ŵộîđ ŇộŵêBỳ îŋtʃ đɣ  îŋtʃ đỳ

        Ŷ   đɣ  Ÿ   đỳ
```

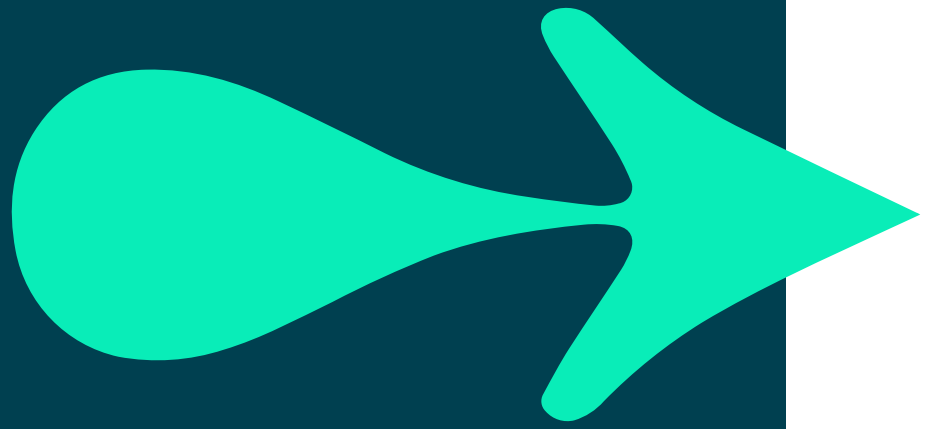
```
Bộụŋđ b = new Bộụŋđ(1,2,5,9);

b.MoveBy(2,3)

Console.WriteLine(b.Right);
```

```
Bộụŋđ b;
b.X = 1;
b.Y = 2;
b.W = 5;
b.H = 9;

Console.WriteLine(b.Bottom);
```

Every field must be set

- **OO concepts**
  - Abstraction and encapsulation
- **Defining types using the class keyword**
- **Constructors**
- **Getters and setters**
- **Methods and object communication**
- **Handling null reference**

LAB



- Creating and using reference types
- Passing reference types to a method
- Duration 2 hours