OO Programming

C# Programming

In this chapter we will learn ...

- What is Object Oriented Programming
- The four Pillars of OO Programming which are
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Agenda

- Review Last Topic
- Introducing Object Oriented Programming
- The Four Pillars of OO
- Understanding the four pillars - 101

Review

What is ...

- an object?
- a class>

What's the difference?

Introducing Object Oriented Design and Programming

How do we manage the complexity of a complex system?

(an) Answer

We break it down into smaller pieces

- Each piece should be easier to understand
- Each of the pieces can then be linked together and made to work with each other?
- Each piece can be verified/tested

Decomposing Complexity

To break a complex system into smaller pieces we could write

- Modules containing monolithic code - ???
- Modules containing functions - Functional approach
- Classes defining data and methods - OO Approach

OO Analysis and Design

Traditional Approach

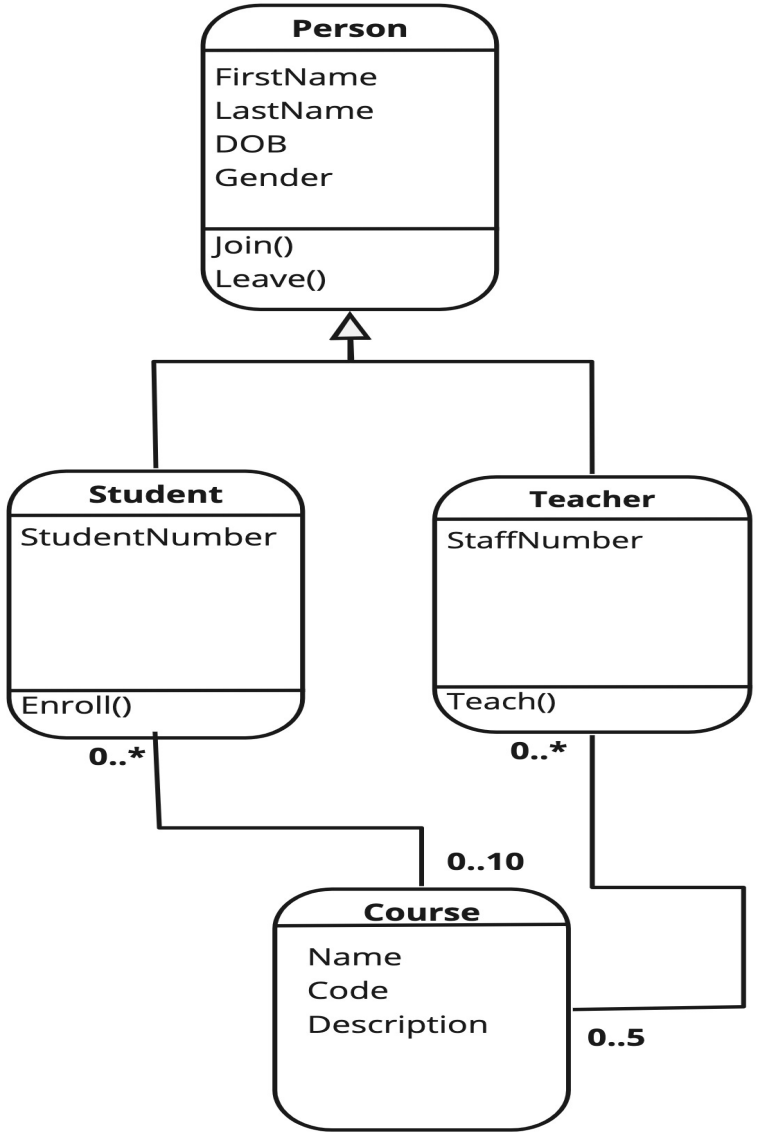
- Produce UML Models of domain
  - Class Diagrams
  - State Transition Diagrams
  - Object Sequence Diagrams
- Usually done before development commences

Agile Approach

- No upfront design (No actual value to shippable product)
- Design emerges over iteration (UML may be used to document design)

Modeling a problem with class diagrams

- Identify the "nouns" in a problem statement
  - Nouns are modeled as classes
- Determine relationships between classes
- Identify commonality
  - Can you make generalizations
  - Identify specializations
  - Use inheritance

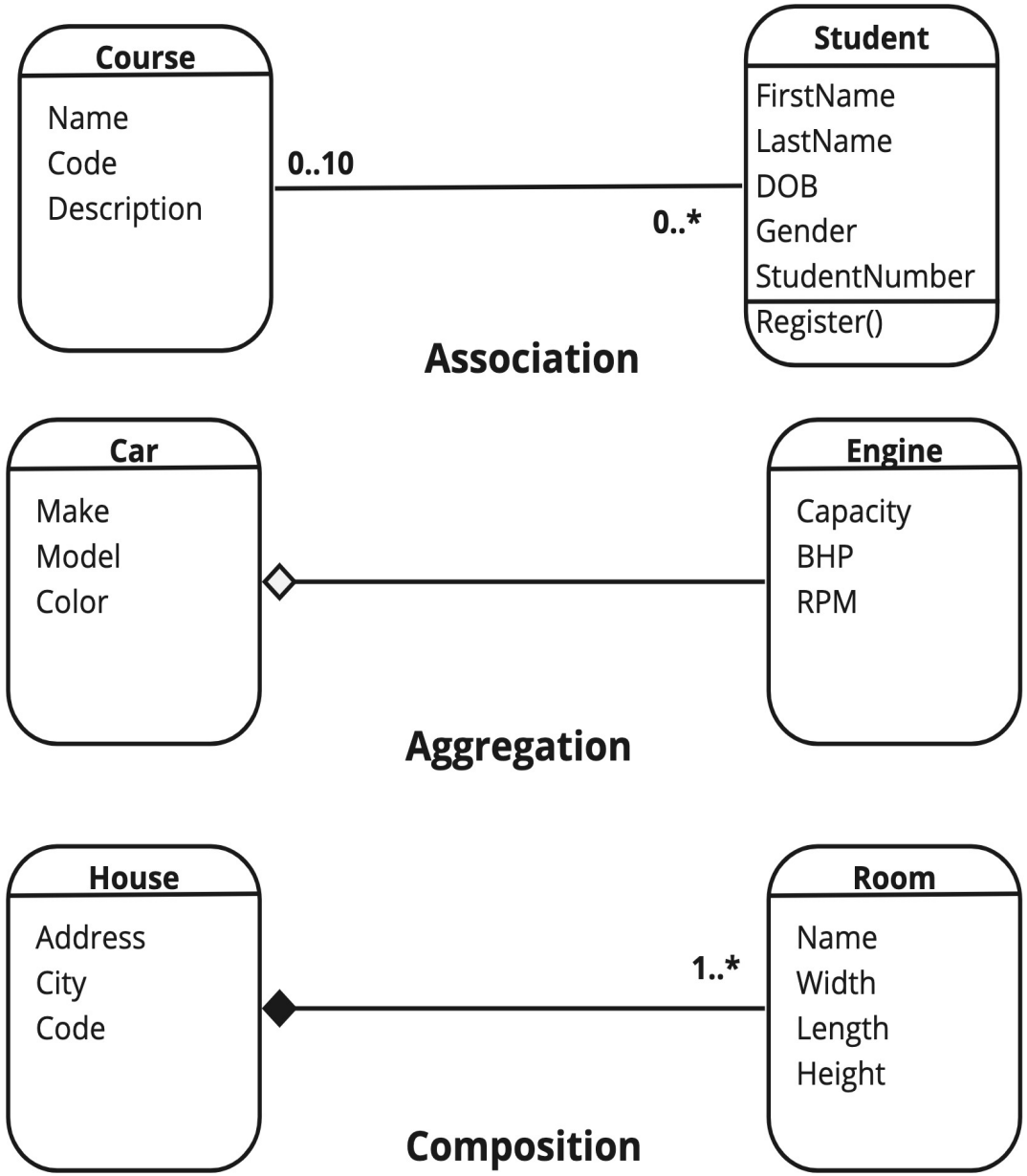


Associations

Define object relationships

Three types

- Association - "Knows of"
- Aggregation - "Part of"
- Composition - "Part of"



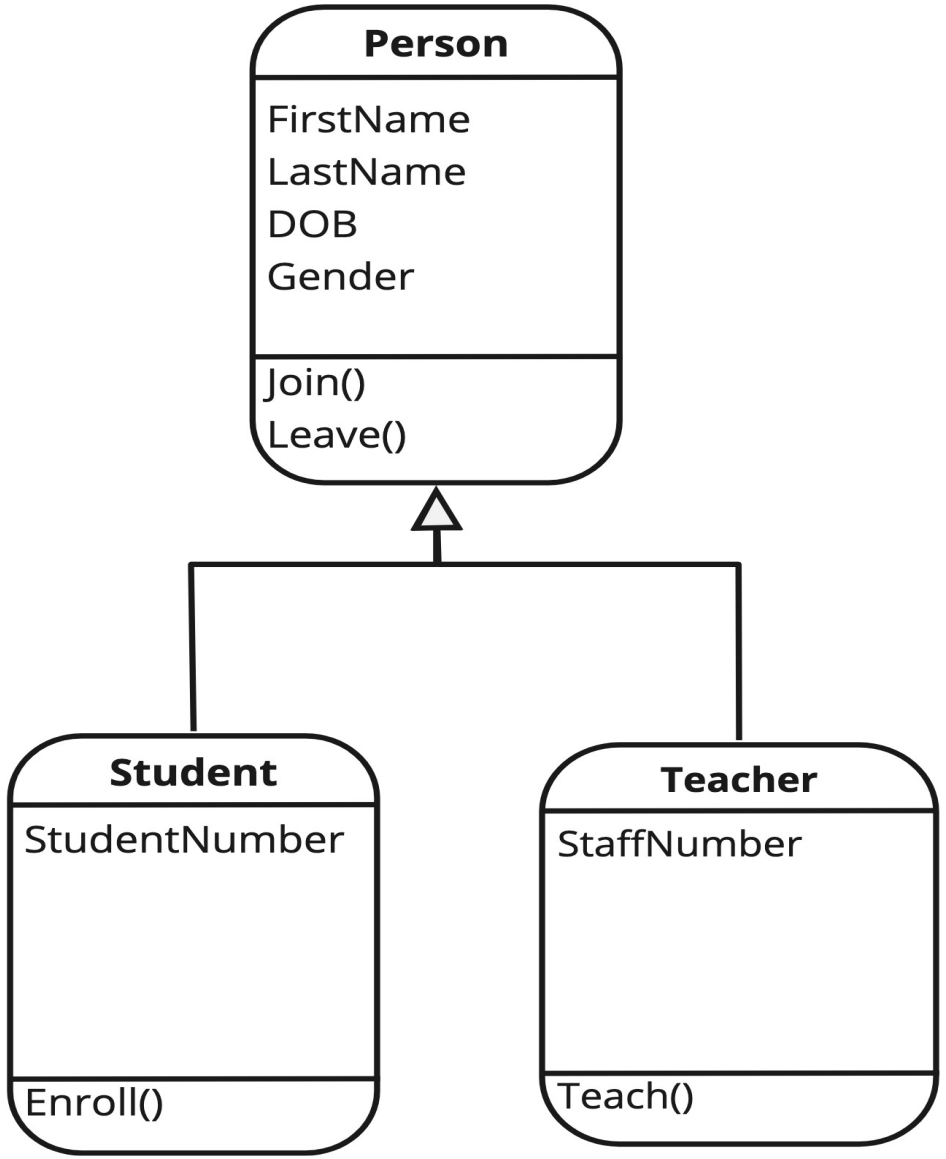
Inheritance

Generalizations

- Define Commonality across types

Specializations

- Extend a base type
  - Add new behavior
  - Override existing behavior



Aggregation

Association

- Applies to types that are
  - Related
  - Utilize the services of

Aggregation

Interface

Defines capability without defining implementation

A document could be:

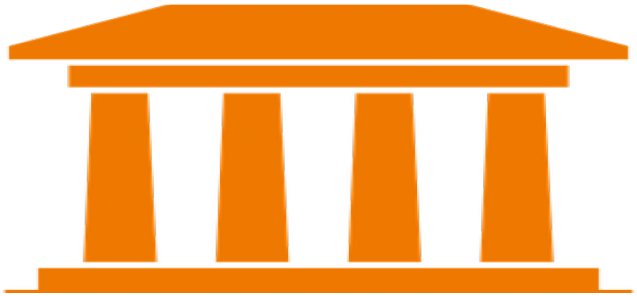
- **SpellCheckable** and
- **Printable** at the same time

That means the document has the capability and therefore the ability to be

- Printed and
- Spell Checked

Interfaces define the **methods** and **properties** that a document must implement for it to have both capabilities

Principles of OO



The Four Pillars

- Abstraction - Hide complexity of implementation
- Encapsulation - Hide and protect data
- Inheritance - Identify commonality/specialization and promote re-use
- Polymorphism - Write code that works with objects based on capability

Creating a Good OO Design

Ideally a good OO design should produce an object model that is

- Loosely Coupled
- Easy to Test
- Flexible enough to change
- Able to fundamentally switch implementation

Design is said to be SOLID

Encapsulation

Encapsulation

Purpose

- Hide a type's information from direct manipulation
- Allow only controlled access to a type's data

Benefit

- Data inside objects changes in controlled and predictable ways
- Data changes can be validated
- Data changes can be easily detected and intercepted

Information Hiding

Consider the following

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

```
Car myCar = new Car();
myCar.speed = 50; // Accessible because it is public
myCar.make = "Ford"; // Not Accessible because it is private
... ?
```

Controlling Access to Data

Using Car as an example

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

- We could make Car go faster than any car has ever gone!

```
Car myCar = new Car();
myCar.speed = 50000; // 50000 MPH is faster than the fastest car ever
```

PROBLEM

Allowing Direct Access to a types data creates problems

- We can't control how that data is manipulated
- We can't validate inputs
- We can't ensure calculations are performed correctly
- We can't ensure processes are followed

It's a free for all!

Encapsulation

Encapsulation is a form of information hiding

- Make data private inside a type
- Provide functions/properties to allow controlled access to the data

Reading private data

Private data can be retrieved using a public getter function

```
class Car {
    private int speed;
    public int GetSpeed(){
        return speed;
    }
}
```

Modifying private data

Public Setter functions can control the change of hidden data

```
class Car {
    public const int MAX_SPEED = 120;
    private int speed;
    ...
    public int SetSpeed(int value){
        if (value <= MAX_SPEED) {
            speed = value;
        }
    }
}
```

Properties

C# allows for the use of properties

- Appear to be simple fields
- Implemented using Getter/Setter functions
- Use private backing fields to store data

Properties are an alternative to getter/setter functions

Property - Example

Defining a Read Property

```
class Car {
    public const int MAX_SPEED = 120;
    private int speed;
    public int Speed{
        get {
            return speed;
        }
    }
}
```

Property - Example

Defining a Write Property

```
class Car {
    public const int MAX_SPEED = 120;
    private int speed;
    public int Speed{
        set {
            if (value <= MAX_SPEED) {
                speed = value;
            }
        }
    }
}
```

- Note the value parameter isn't formally defined
- value is an implicit parameter of a setter

Property Expressions

Properties can be condensed

```
class Car {
    public const int MAX_SPEED = 120;
    private int speed;
    public int Speed {
        get => speed;
        set => if (value < MAX_SPEED) speed = value;
    }
}
```

Varying Accessibility

Read/Write accessors can have different accessibility settings

- getter is public
- setter is private

```
class Car {
    public const int MAX_SPEED = 120;
    private int speed;
    public int Speed {
        get => speed;
        private set => if (value < MAX_SPEED) speed = value;
    }
}
```

Inheritance

Question

What are the major benefits of Inheritance?

Inheritance

How do you understand it?

- A child inherits some features of its parents
- A beneficiary inherits some assets from an estate
- A type inherits some features of its parent type

In each case

- "Features" are being inherited from parent to child

Inheriting Features

The ability to inherit features is useful to programmers

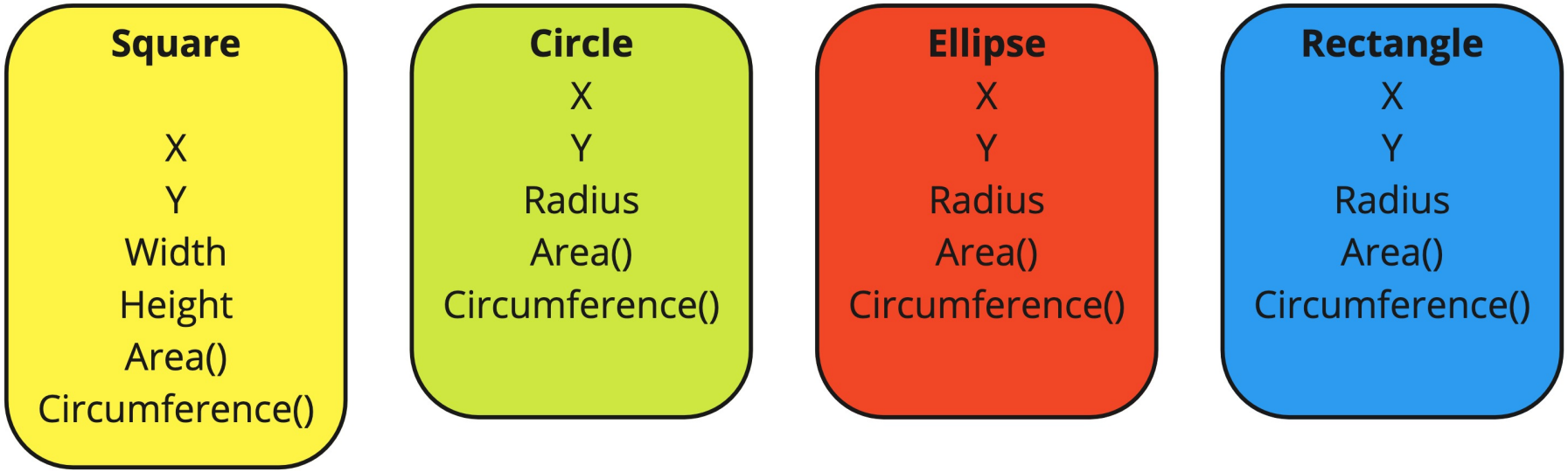
- We could define general features in a parent type (Generalization)
- Inherit the general features into one or more child types
- Add extra features specific to each child (Specialization)

Benefit

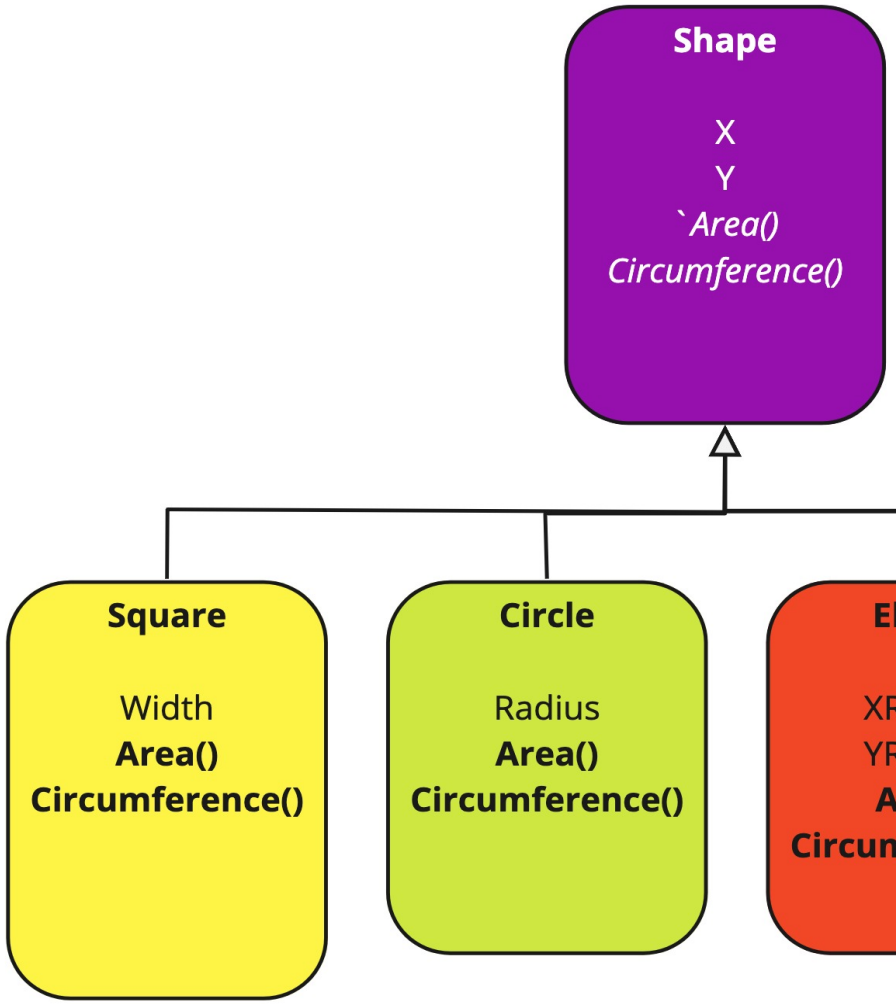
- We are writing less code
- We are reusing code

Example - Shapes have an Origin, Size, Area

What do they have in common?



Creating a Generalization - Shape



Abstraction/Generalization

Area/Circumference are needed for all Shapes but implementation is specific to each

Specialization

Allows new features to be added and base features to be overridden to vary/specialize behavior of derived types

C# Inheritance

Syntax

```
### C# supports single inheritance
class <Derived Type Name> : <Base Type Name> {

}
```

Implementing Inheritance

Here Rectangle inherits from Shape with a final definition of

- X, Y
- Width, Height

```
class Shape { // Base Class
    public int X {get;set;}
    public int Y {get;set;}
}
```

```
class Rectangle: Shape { // Derived Class
    public int Width {get;set;}
    public int Height {get;set;}
}
```

Inheritance and constructors

Here we have a Shape with a constructor

```
class Shape { // Base Class
    public int X {get;init;}
    public int Y {get;init;}
    public Shape(int x, int y){
        X = x;
        Y = y;
    }
}
```

- Following code initializes an instance

```
Shape myShape = new Shape(10, 5);
```

Potential Problem

What happens when an inheriting type inherits from Shape?

- Constructors aren't inherited

```
class Square: Shape
{

}
```

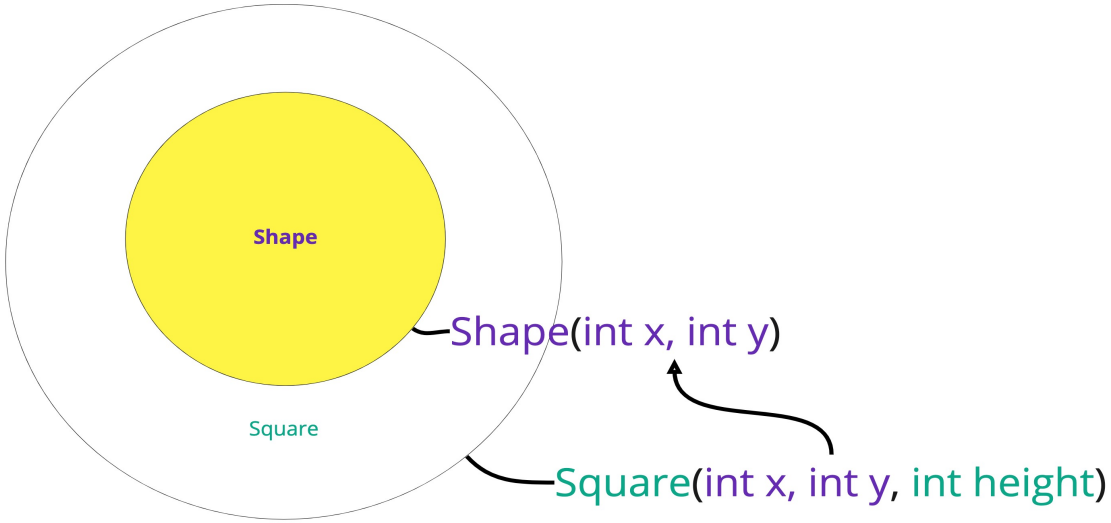
Answer

It's a problem!!!!!

Single Inheritance

To create a derived type

- Its base type has to be constructed first



- A base type constructor must be satisfied

Calling the Base Constructor from the Derived class

Here we

- Add a constructor to Square
- Call the base constructor directly using :base (x, y) to initialize the base
- Initialize Width for Square constructor

```
class Square: Shape
{
    public Square(int x, int y, int width): base(x, y) {
        Width = width;
    }
}
```

Providing a Blank Constructor

If the base class has a parameter-less constructor

- Parameter-less constructor will be used automatically
- Explicit Constructor chaining is unnecessary

```
class Shape {
    public Shape() {} // Parameterless Constructor Added
    public Shape(int x, int y){
        X = x;
        Y = y;
    }
}
class Square: Shape {  // Square can construct Shape using parameterless constructor
}
```

Extending Classes

Classes can be extended by adding new features to the derived class

```
class Shape { // Base Class
    public int X {get;set;}
    public int Y {get;set;}
}
```

Here Shape is extended by adding property Height

```
class Square: Shape { // Derived Class
    public int X {get;set;}
    public int Y {get;set;}
    public int Height {get;set;}
}
```

Allowing base class features to be overridden

Derived classes can override methods/properties declared in base as

- abstract - must override in derived class
- virtual - can override or be inherited by derived class

```
abstract class Shape { // Base Class
    public int X {get;set;}
    public int Y {get;set;}
    public abstract int Area {get;}
    public abstract int Circumference {get;}
    public virtual Point Center => new Point(X, Y);
}
```

Overriding base class features

We use override keyword to override properties/methods in base class

```
class Rectangle: Shape { // Derived Class
    public int Width {get;set;}
    public int Height {get;set;}
    public override int Area => Width * Height;
    public override int Circumference => Width * 2 + Height * 2;
}
```

- X and Y properties are inherited from Shape
- Virtual **Center** property is inherited **but** we could override this
- Shape is extended by adding **Width** and **Height** properties
- Concrete implementations for abstract Area and Circumference properties are added

So ...

Inheritance allows us to

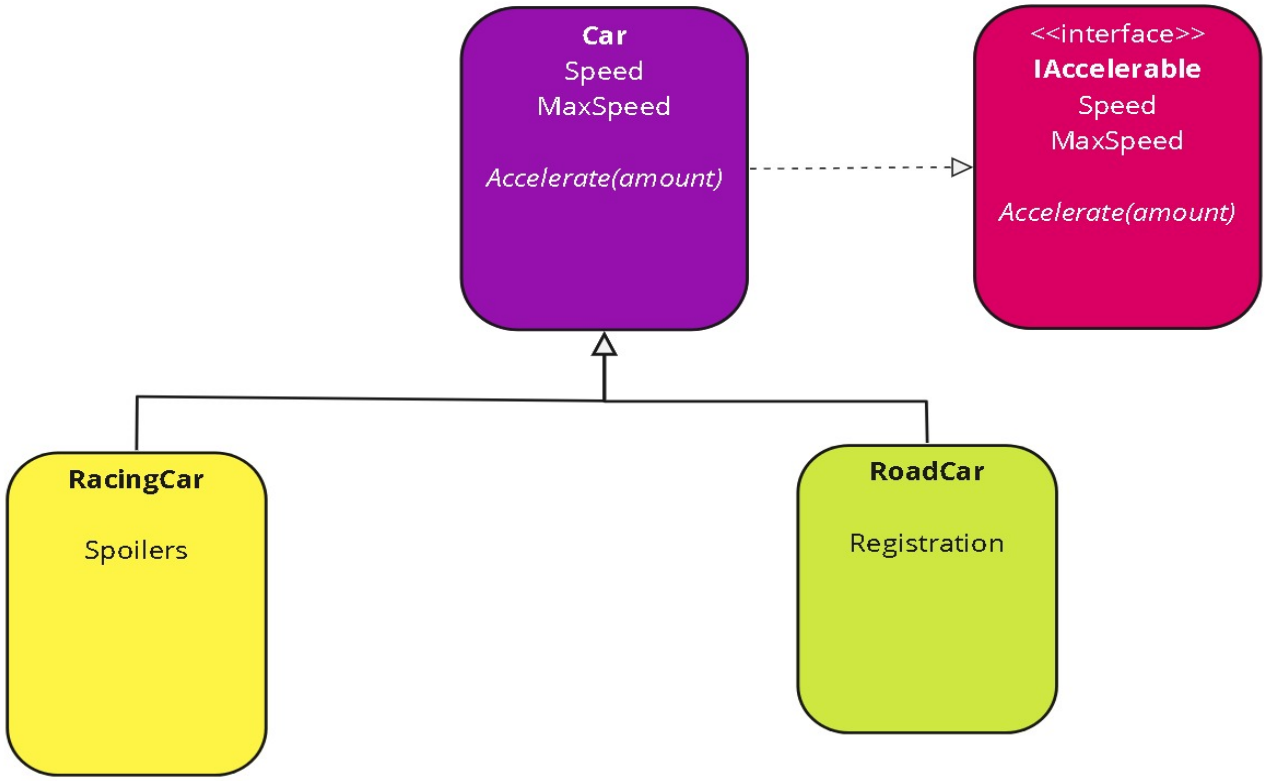
- Inherit and re-use features in a base class
- Extend features in a base class by adding new features to the derived class
- Override abstract or virtual features of the base class
    - Allows us to vary the behavior of the same method across multiple derived classes
    - Facilitates Polymorphism

Polymorphism

Polymorphism

Literally means Multiple Forms

Let's image the following model



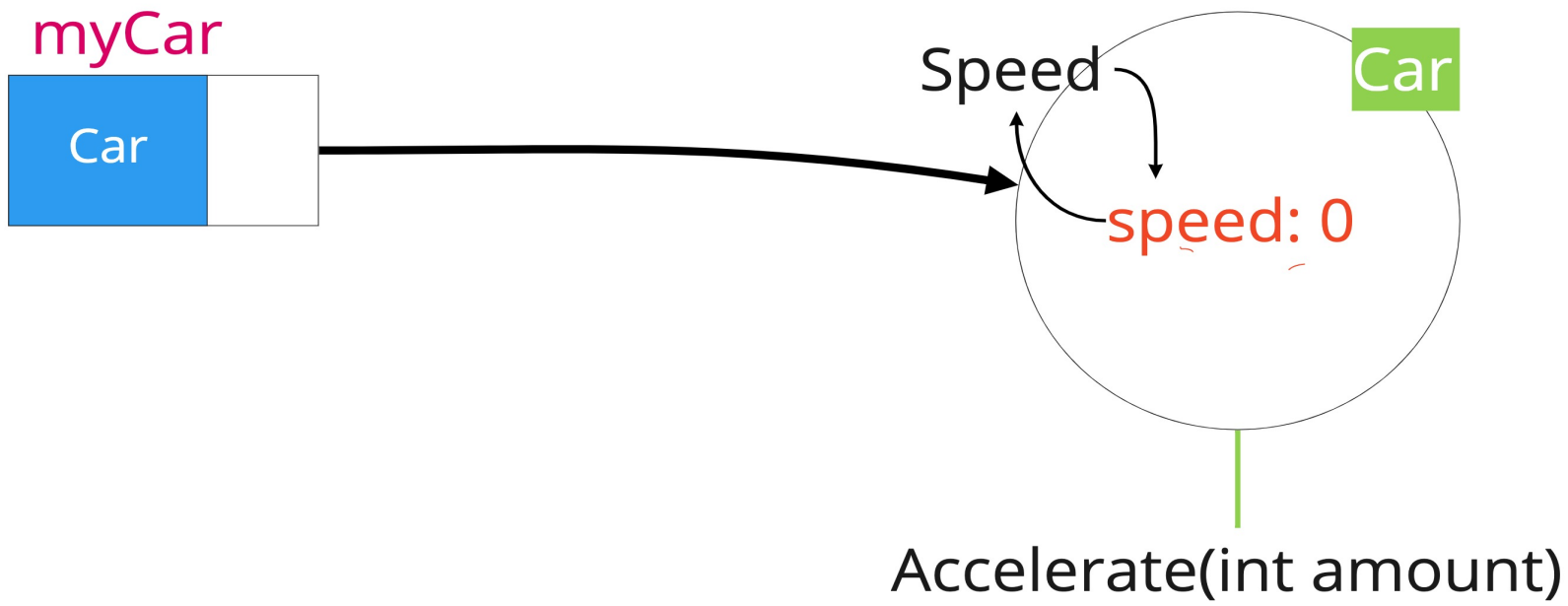
Polymorphism Explained

```
class Car {
    public int Speed {get;protected set;}
    public virtual int MaxSpeed {get => 120}
    public virtual void Accelerate(int amount = 5) {
        Thread.Sleep(80 * amount);
        Speed += amount;
    }
}
```

Creating a Car object

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

- Car reference points to Car object



Extending Car

RacingCar is a kind of Car but it also has

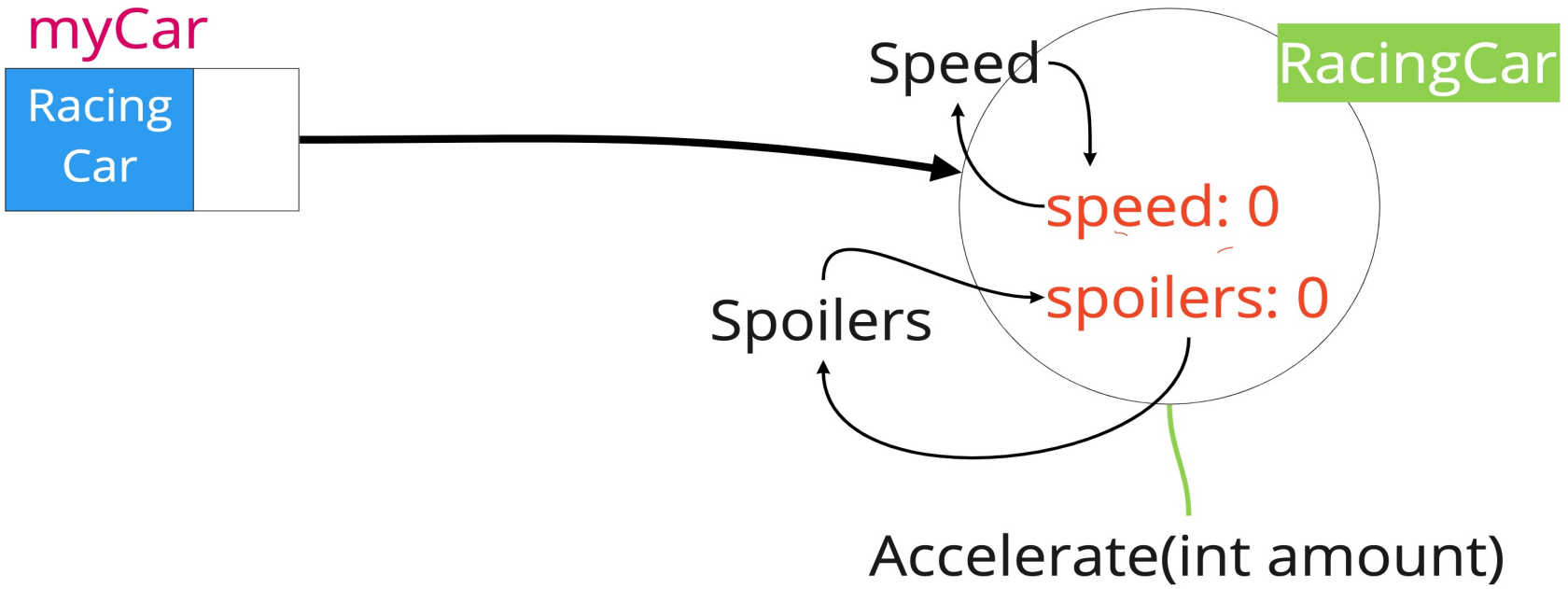
- Spoilers property

```
class RacingCar: Car {
    public int Spoilers {get;set;}
}
```

Creating a RacingCar object

```
RacingCar myCar = new RacingCar();
```

- RacingCar reference points to RacingCar object



Reference Substitution

A base class reference variable can point to

- Any derived object in inheritance hierarchy

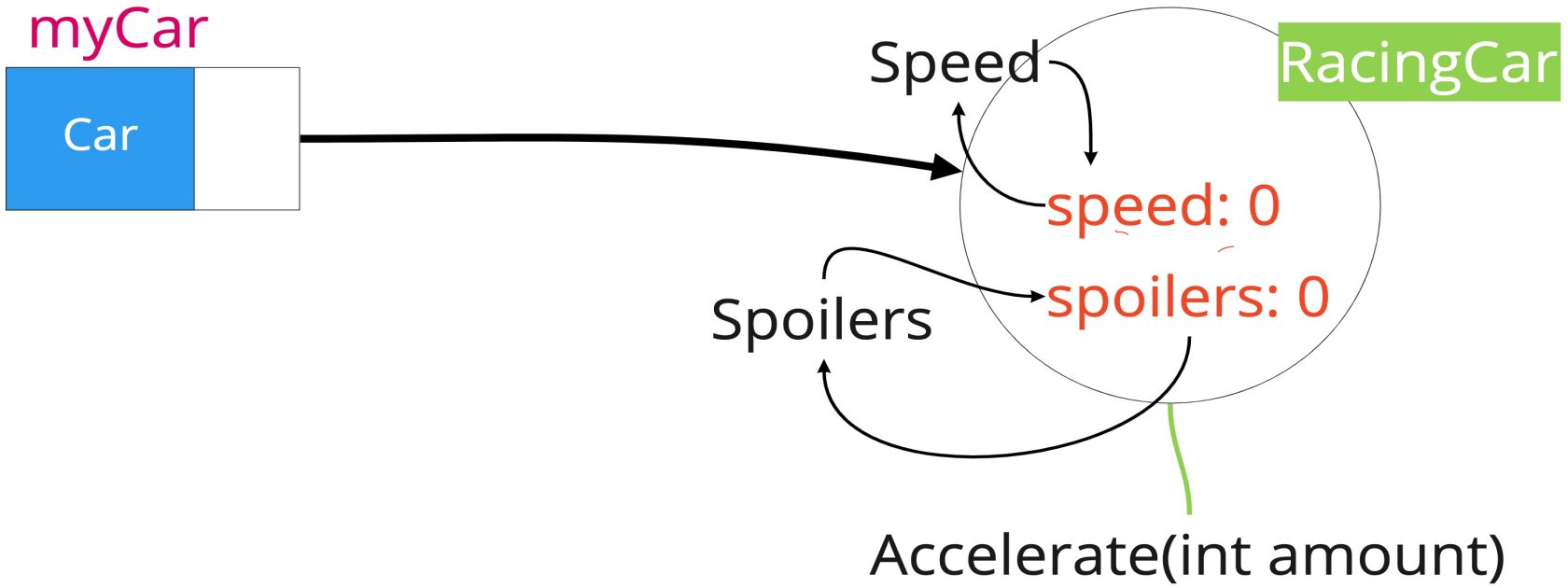
That means we can do this

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

Reference Substitution - Under the hood

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

- Car reference points to RacingCar object



Why is substitution useful?

We can write sub systems targeting the base class

- e.g. A function expecting a base class parameter

Use method overriding in derived classes to vary behavior

- Derived classes override certain methods/properties

Substitute base class for derived instances

- Pass an instance of a derived class rather than the base class
- Result is polymorphism

Given

```
void AccelerateToMaxSpeed(Car item) {
    while(item.Speed < item.MaxSpeed) {
        item.Accelerate();
    }
}
```

Substitution allows us to pass to AccelerateToMaxSpeed a

- Car reference or ..
- Reference of any type inheriting from Car

```
AccelerateToMaxSpeed(new Car());
AccelerateToMaxSpeed(new RacingCar());
```

So what's the big deal?

At the moment Car and RacingCar will both Accelerate the same way

- Accelerate method is inherited from Car

But .. It's declared as virtual

- virtual means it can be overridden

Let's override Accelerate(..)

```
class Car {
    ...
    public virtual Accelerate(int amount = 5){
        Thread.Sleep(80 * amount);
        Speed += amount;
    }
}
```

- Reducing the delay will make it appear to accelerate faster

```
class RacingCar: Car {
    ...
    public override Accelerate(int amount = 5){
        Thread.Sleep(40 * amount);
        Speed += amount;
    }
}
```

Same Code - Different Object

Now let's Accelerate different Cars ...

```
AccelerateToMaxSpeed(new Car());
AccelerateToMaxSpeed(new RacingCar());
```

Result

RacingCar accelerates faster than Car

- Substitution allows us to pass any kind of Car to AccelerateToMaxSpeed
- override let's us modify the behavior of certain features

Polymorphism

The ability to define multiple object forms that

- Share a common set of behaviors
- Implement their behaviors in their own way

Polymorphism helps us to

- Increase the flexibility and reuse of the code we write
- Decouple the code we write from dependence of data type

Abstraction

Discuss

What is Abstraction?

Imagine a Duck

Why is a Duck like abstraction?

Abstraction

Abstraction is about hiding complexity by ...

- Hiding the details of implementation
- Simplifying the external surface
    - Providing only those methods and properties absolutely needed

Abstraction is about ease of change

- We should be able to change how the type is implemented
- We shouldn't need to change how we use it

What does it mean to be a ...

Car

What are the bare essentials that define Car in our domain?

- Speed, RPM, FuelLevel, VID ... ?
- Accelerate(), Brake(), Start(), Stop() ... ?

The Job

Condense the definition down to bare essentials

Hide the complexity

- How do we generate power? … Hide it
- How do we store fuel? … Hide it
- How do we drive the wheels? … Not your problem

We need to define a simple interface that

- Allows the user of the type to operate the Car regardless of its implementation

Why might this implementation be a problem?

```
class Car {
    public int Speed {get;private set;}
    public Accelerate(int amount) {
        int requiredSpeed = Speed + amount;
        while(Speed < requiredSpeed) {
            IncreaseFuelFlow(5);
            Thread.Sleep(500);
        }
    }
    public void IncreaseFuelFlow(int volumeIncrease) {
        // Increase Fuel Flow
        // Leads to Speed increasing ... trust me :)
    }
}
```

Abstracting Car

We can change the speed of the Car by calling public methods

- Accelerate() or
- IncreaseFuelFlow()

But ...

- IncreaseFuelFlow() assumes a fossil fuel based Car
- What happens if we want to change the implementation to be electric?

So create an abstract public interface

Hiding Implementation

If we hide implementation we can change it later

```
class Car {
    public int Speed {get;private set;}
    public Accelerate(int amount) {
        int requiredSpeed = Speed + amount;
        while(Speed < requiredSpeed) {
            IncreaseFuelFlow(5);
            Thread.Sleep(500);
        }
    }
    private void IncreaseFuelFlow(int volumeIncrease) {
        ...
    }
}
```

To change implementation for an Electric Car

- Maintain the public interface
- Change the private implementation

```
class Car {
    public int Speed {get;private set;}
    public Accelerate(int amount) {
        ...
        while(Speed < requiredSpeed) {
            IncreaseElectricCurrent(5);
            ...
        }
    }
    private void IncreaseElectricCurrent(int current) {}
}
```

So ...

The public face of our types should be simple (an abstraction) but ...

- Under the hood they are implementing complexity via
  - Private function
  - Interactions with other types

We should be able to change the implementation of our type without

- Changing the public interface
- Changing how it is used within our application
- Unit Test cases should still hold true

Abstract Data Types

Sometimes an abstraction is

- Too abstract to exist as an object in its own right
    - Useful only as a template for defining other types

C# Allows for the creation of abstract type using abstract keyword

- Defines a type that can't be used to create objects. Does support inheritance

```
abstract class Car {
    public int Speed {get;private set;}
}
Car myCar = new Car(); // COMPILE ERROR!!!
```

Abstract Classes

Abstract classes allow abstract methods and properties

- Requires the inheriting class to implement

Code below requires any inheriting type to fully implement Accelerate(..)

- Note Accelerate has no code block defined { .. }

```
abstract class Car {
    public int Speed{get;private set;}
    public abstract void Accelerate(int amount);
}
```

In this chapter we learned ...

- What is Object Oriented Programming
- The four Pillars of OO Programming which are
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism