Towards the .NET Junior Developer

The extremely solid course

Lesson 3

Object-Oriented Programming in C#

Part I

Agenda

- OOP Basics
  - Encapsulation
  - Inheritance
  - Polymorphism
- Books of the day
- Links of the day
- Hometask

OOP Basics

Encapsulation

“**Encapsulation** refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components”

Wikipedia

Properties

```
// Get/Set props
0 references
public string? GetSetAutoProp { get; set; }


private string _getSetProp = string.Empty;
0 references
public string GetSetProp
{
    get
    {
        return _getSetProp;
    }
    set
    {
        _getSetProp = value;
    }
}

// Get only props
0 references
public string? GetOnlyAutoProp { get; }
```

```
var props = new Properties();

props.GetSetProp = "Can set it";
Console.WriteLine(props.GetSetProp);
```

Properties

```
// Get/Set props with check
private string _getSetPropWithCheck = string.Empty;
0 references
public string GetSetPropWithCheck
{
    get
    {
        return _getSetPropWithCheck;
    }
    set
    {
        if (_getSetPropWithCheck == null)
        {
            _getSetPropWithCheck = value;
        }
    }
}
```

Properties

```
// Get only props
0 references
public string? GetOnlyAutoProp { get; }


0 references
public string? GetOnlyLambdaStyleProp => string.Empty;


private string _getOnlyProp = string.Empty;
0 references
public string GetOnlyProp
{
    get
    {
        return _getOnlyProp;
    }
    set
    {
        _getOnlyProp = value;
    }
}
```

```
props.GetOnlyAutoProp = "Can't set it";
Console.WriteLine(props.GetOnlyAutoProp);

props.GetOnlyLambdaStyleProp = "Can't set it";
Console.WriteLine(props.GetOnlyLambdaStyleProp);

props.GetOnlyProp = "Can't set it";
Console.WriteLine(props.GetOnlyProp);
```

Properties

```
// Set only props
private string _setOnlyProp = string.Empty;
0 references
public string SetOnlyProp
{
    set
    {
        _setOnlyProp = value;
    }
}
```

```
props.SetOnlyProp = "Can set it";
Console.WriteLine(props.SetOnlyProp);
```

Properties

```
// Get/private Set props
0 references
public string? GetPrivateSetAutoProp { get; private set; }


private string _getPrivateSetProp = string.Empty;
0 references
public string GetPrivateSetProp
{
    get
    {
        return _getPrivateSetProp;
    }
    private set
    {
        _getPrivateSetProp = value;
    }
}
```

```
props.GetPrivateSetProp = "Can't set it here";
Console.WriteLine(props.GetPrivateSetProp);
```

Properties

```
// Get/init props
0 references
public string? GetInitAutoProp { get; init; }


props.GetInitAutoProp = "Can't set it here";
Console.WriteLine(props.GetInitAutoProp);

var props2 = new Properties { GetInitAutoProp = "But can set here" };

var props3 = props2 with { GetInitAutoProp = "Updated value" };
```

Encapsulation demo

OOP Basics

Inheritance

“**Inheritance** is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation”

Wikipedia

Inheritance

Pepperoni

derived class

is

Pizza

base class

Inheritance

Pizza

base class

~~is~~

Pepperoni

derived class

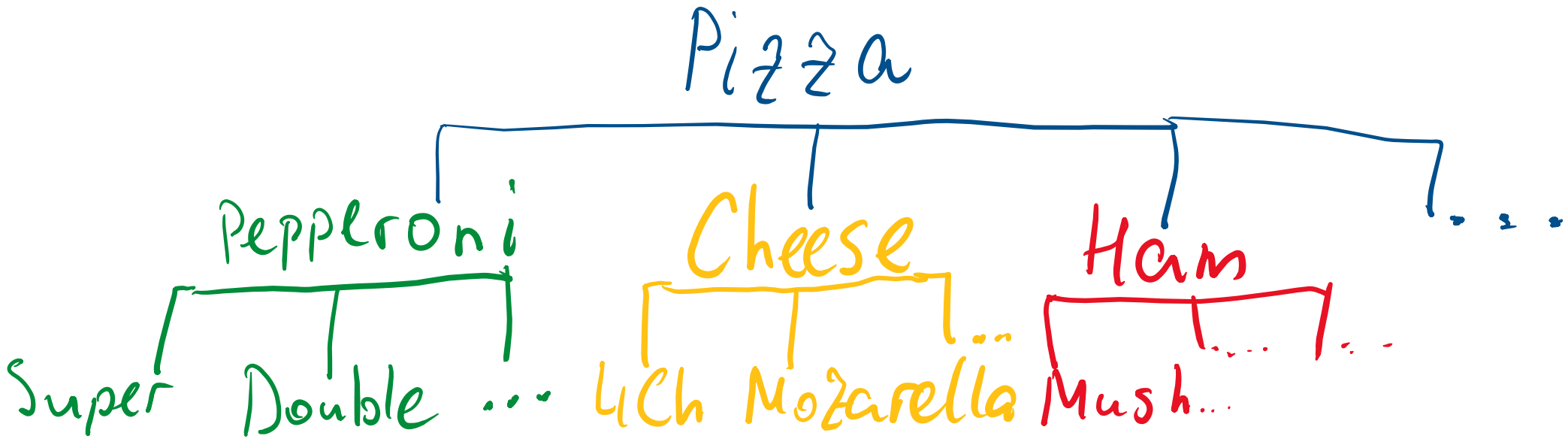
Inheritance

```
public class Pizza
{
    1 reference
    public Pizza(string name, PizzaSize? size = PizzaSize.Medium)
    {
        Name = name;
        Size = size;
    }

    1 reference
    public string Name { get; }

    1 reference
    public PizzaSize? Size { get; }

    2 references
    public Ingredient[] Ingredients { get; protected set; } = Array.Empty<Ingredient>();

    1 reference
    public virtual void PrintRecipe()
    {
        Console.WriteLine("I'm just a pizza draft!");
    }
}
```

Inheritance

```
public class Pepperoni : Pizza
{
    private const string NAME = "Pepperoni";

    0 references
    public Pepperoni(PizzaSize? size) : base(NAME, size)
    {
        Ingredients = new Ingredient[]
        {
            new ("dough"),
            new ("cheese"),
            new ("pepperoni"),
        };
    }

    1 reference
    public override void PrintRecipe()
    {
        Console.WriteLine($"Pizza: {NAME}");
        foreach(var ingredient in Ingredients)
        {
            Console.WriteLine($"\t-{ingredient}");
        }
    }
}
```

Inheritance



1 – N

Inheritance – abstract classes

```
public abstract class WiFiConsumer
{
    3 references
    public abstract string Name { get; }

    1 reference
    protected string FirmwareVersion => "1.0.2";

    2 references
    protected virtual int Downloaded { get; }

    2 references
    protected virtual int Uploaded { get; }

    2 references
    protected abstract void ReportTrafficStats();

    2 references
    public virtual void PrintHardwareInfo()
    {
        Console.WriteLine($"Firmware ver.: {FirmwareVersion}");
    }
}
```

Can't create WiFiConsumer directly

No method body

Inheritance – abstract classes

```
0 references
public class Smartphone : WiFiConsumer
{
    1 reference
    public override string Name => "Smartphone";

    2 references
    protected override int Downloaded => new Random().Next(1024);

    2 references
    protected override int Uploaded => new Random().Next(256);

    1 reference
    protected override void ReportTrafficStats()
    {
        Console.WriteLine($"D: {Downloaded} Mb via WiFi (excl. LTE)/ U: {Uploaded} Mb via WiFi (excl. LTE)");
    }
}
```

Have to override

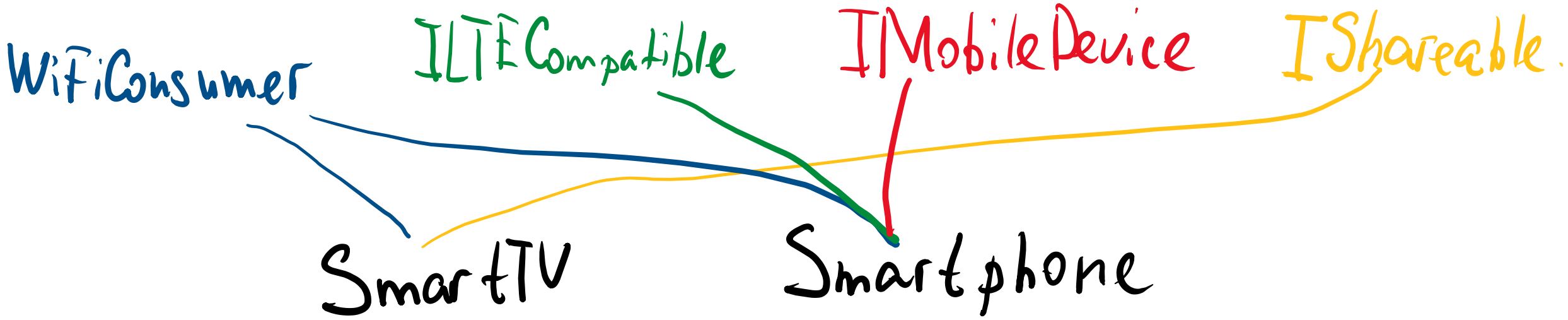
Have to override

Inheritance – abstract classes

```
public class SmartTV : WiFiConsumer
{
    2 references
    public override string Name => "SmartTV";

    2 references
    protected override int Downloaded => 333;

    2 references
    protected override int Uploaded => 111;

    1 reference
    protected override void ReportTrafficStats()
    {
        Console.WriteLine($"Downloaded {Downloaded} Mb/ Uploaded {Uploaded} Mb");
    }

    2 references
    public override void PrintHardwareInfo()
    {
        base.PrintHardwareInfo();
        Console.WriteLine($"{Name} Model G2022-S300LT");
    }
}
```

Optionally overridable (virtual)

Inheritance – interfaces



N – N

Inheritance - interfaces

```
internal interface ILTECompatible
{
    1 reference
    bool IsConnected { get; }
    1 reference
    bool SearchForBaseStation();
}
```

```
public class Smartphone : WiFiConsumer, ILTECompatible
{
    1 reference
    public override string Name => "Smartphone";

    1 reference
    public bool IsConnected => SearchForBaseStation();

    2 references
    protected override int Downloaded => new Random().Next(1024);

    2 references
    protected override int Uploaded => new Random().Next(256);

    2 references
    public bool SearchForBaseStation()
    {
        return true;
    }

    1 reference
    protected override void ReportTrafficStats()
    {
        Console.WriteLine($"D: {Downloaded} Mb via WiFi (excl. LT
    }
}
```

Inheritance – abstract classes vs interfaces

Prefer **abstract class**, if classes are linked by “A is B” relationships and have no more than one parent.

Prefer **interface**, if you have a deal with the behavior aspects (contracts) – “A should be able to …”.

Inheritance demo

It’s coffee time!

OOP Basics

Polymorphism

“**Polymorphism** is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types”

Wikipedia

Polymorphism - methods

```
public class Pepperoni : Pizza
{
    private const string NAME = "Pepperoni";

    0 references
    public Pepperoni(PizzaSize? size) : base(NAME, size)
    {
        Ingredients = new Ingredient[]
        {
            new ("dough"),
            new ("cheese"),
            new ("pepperoni"),
        };
    }

    1 reference
    public override void PrintRecipe()
    {
        Console.WriteLine($"Pizza: {NAME}");
        foreach(var ingredient in Ingredients)
        {
            Console.WriteLine($"\t-{ingredient}");
        }
    }
}
```

Polymorphism - methods

```
var basePizza = new Pizza("Useless name", PizzaSize.Large);
basePizza.PrintRecipe();

var pepperoni = new Pepperoni(PizzaSize.Small);
pepperoni.PrintRecipe();
```

```
I'm just a pizza draft!
Pizza: Pepperoni, size: Small
        -Ingredient { Name = dough }
        -Ingredient { Name = cheese }
        -Ingredient { Name = pepperoni }
```

override – sets the derived class own implementation of method (property)

Polymorphism - methods

```
public new void PrintRecipe()
{
    Console.WriteLine($"Pizza: {NAME}, size: {Size}");
    foreach(var ingredient in Ingredients)
    {
        Console.WriteLine($"\t-{ingredient}");
    }
}
```

```
I'm just a pizza draft!
Pizza: Pepperoni, size: Small
        -Ingredient { Name = dough }
        -Ingredient { Name = cheese }
        -Ingredient { Name = pepperoni }
```

new – creates a new implementation of method (property), saves the parent's one

Polymorphism - methods

```
var basePizza = new Pizza("Useless name", PizzaSize.Large);
basePizza.PrintRecipe();

var pepperoni = new Pepperoni(PizzaSize.Small);
pepperoni.PrintRecipe();

var fakePepperoni = (Pizza)new Pepperoni(PizzaSize.Small);
fakePepperoni.PrintRecipe();
```

```
I'm just a pizza draft!
Pizza: Pepperoni, size: Small
        -Ingredient { Name = dough }
        -Ingredient { Name = cheese }
        -Ingredient { Name = pepperoni }
I'm just a pizza draft!
```

Calls parent’s implementation of method via explicit casting to the base type

Polymorphism - methods

```
public void PrintRecipe()
{
    Console.WriteLine($"Pizza: {NAME}, size: {Size}");
    foreach(var ingredient in Ingredients)
    {
        Console.WriteLine($"\t-{ingredient}");
    }
}
```

```
I'm just a pizza draft!
Pizza: Pepperoni, size: Small
        -Ingredient { Name = dough }
        -Ingredient { Name = cheese }
        -Ingredient { Name = pepperoni }
```

without any modifier – hides the parent's implementation, leads to the warning

Polymorphism – signatures

```
public void DealWithPolymorphism()
{
    Console.WriteLine("I'm a polymorphic method");
}

0 references
public void DealWithPolymorphism(string message)
{
    Console.WriteLine($"I'm a polymorphic method too and have a message for you: {message}");
}

0 references
public void DealWithPolymorphism(object message)
{
    Console.WriteLine($"I'm a polymorphic method too and have another message for you: {message}");
}

0 references
public void DealWithPolymorphism(int number)
{
    Console.WriteLine($"I'm a polymorphic method too and know a simple number: {number}");
}
```

1. w/o args

2. string arg

3. object arg

4. int arg

Polymorphism – signatures

```
public string DealWithPolymorphism()
{
    Console.WriteLine("I can't be used because the method with the same signature has already been added");
    return string.Empty;
}
```

5. w/o args – already exists!

```
0 references
public int DealWithPolymorphism(int number)
{
    Console.WriteLine("I can't be used because the method with the same signature has already been added");
    return number;
}
```

6. with int arg – already exists!

```
0 references
public long DealWithPolymorphism(int number, string message)
{
    Console.WriteLine("I'm a polymorphic method, and there's no other method with the same signature");
    return number;
}
```

7. with two args – OK

```
0 references
public long DealWithPolymorphism(long number)
{
    Console.WriteLine("I'm a polymorphic method, and there's no other method with the same signature");
    return number;
}
```

8. long is not the same as int – OK!

Polymorphism – operators

```
record Point
{
    2 references
    public int X { get; set; }

    2 references
    public int Y { get; set; }
}
```

```
using Polymorphism;

var point1 = new Point { X = 5, Y = 14 };
var point2 = new Point { X = 7, Y = 3 };

var sumPoint = point1 + point2;

Console.WriteLine($"X={sum
```

(local variable) Point? point2

'point2' is not null here.

CS0019: Operator '+' cannot be applied to operands of type 'Point' and 'Point'

Polymorphism – operators

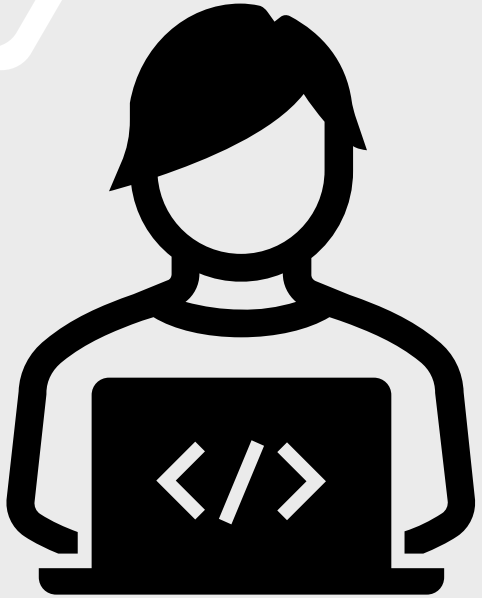
```
record Point
{
    6 references
    public int X { get; set; }

    6 references
    public int Y { get; set; }

    2 references
    public static Point operator +(Point first, Point second) => new Point
    {
        X = first.X + second.X,
        Y = first.Y + second.Y
    };
}
```

```
using Polymorphism;

var point1 = new Point { X = 5, Y = 14 };
var point2 = new Point { X = 7, Y = 3 };

var sumPoint = point1 + point2;

Console.WriteLine($"X={sumPoint.X}, Y={sumPoint.Y}");
```

Microsoft Visual Studio Debug Console

```
X=12, Y=17
```

Let’s practice!



Books of the day

McLaughlin B., Pollice G., West D. - Object-Oriented Analysis And Design

Martin R. – Clean Code

Links of the day

Inheritance in .NET

Diamond inheritance problem – why there's no multiple inheritance in C#?

Polymorphism - MSDN

Polymorphism – good explanation from Jignesh Trivedi (Microsoft MVP)

Hometask

Imagine you're creating a sport goods store. Create a models library that will contain the different types of the sport goods. Don't forget to encapsulate your data correctly. Use inheritance (from base class, from interfaces) where it could be useful. As an example, you can do the things below:

- create the base class for your ierarchy (e.g., StoreItem);
- create the classes-categories that would be derived classes for StoreItem (Wear, Equipment, Food etc.);
- create several items (child classes/records) for every category;
- add the most valuable properties in each item type. The most common ones (like Id, Name, Description) should be placed at the higher
- levels of your items hierarchy;
- use interfaces to mark some items by specific behavior (e.g., interface IHasShelfLife with ExpirationDate property could mark the food that has shelf life).

That’s all for this time!