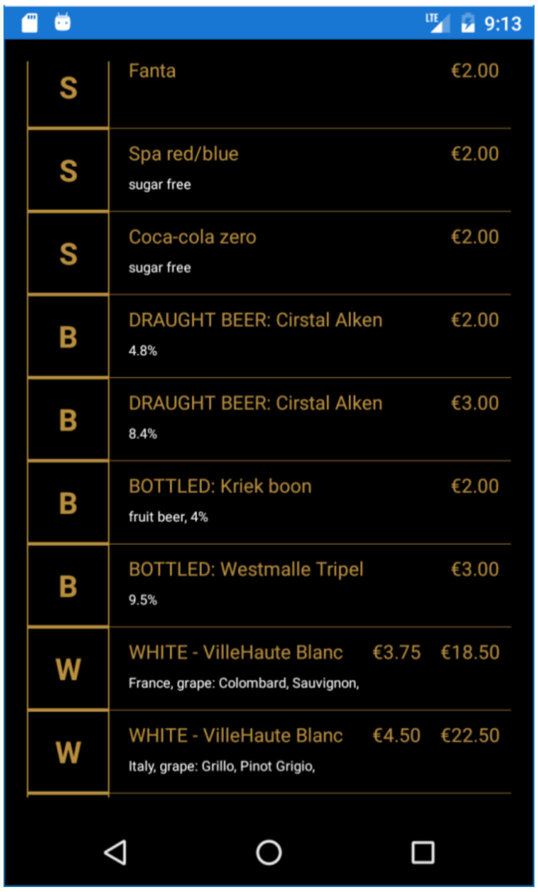
Week4 – Demo Object Oriented Programming

Demo - goal

We wish to create a drink card app. All kinds of Drinks have similar properties and behavior, however, some specific types of drinks will also differ from each other. For example, Wine will have a specific WineType (red, white,…), while other Drinks don't. Also, they might have different serving instructions.

We create a **base class** called 'Drink' that contains common properties such as Name and Price.

The main goal is to create global list that contains all kinds of drinks, which should be automatically displayed using their own specific behavior (or the main behavior if no specific behavior was described).



S	Fanta		€2.00
S	Spa red/blue sugar free		€2.00
S	Coca-cola zero sugar free		€2.00
B	DRAUGHT BEER: Cirstal Alken 4.8%		€2.00
B	DRAUGHT BEER: Cirstal Alken 8.4%		€3.00
B	BOTTLED: Kriek boon fruit beer, 4%		€2.00
B	BOTTLED: Westmalle Tripel 9.5%		€3.00
W	WHITE - VilleHaute Blanc France, grape: Colombard, Sauvignon,	€3.75	€18.50
W	WHITE - VilleHaute Blanc Italy, grape: Grillo, Pinot Grigio,	€4.50	€22.50

Sodas ('S') displaying whether they are sugar free (show nothing if they're not).

Beers ('B') displaying their name preceded by 'bottled' or 'draught beer', and also showing their alcohol %.

Wines ('W') displaying their name preceded by the wine type. The description shows the country of origin.

Replacing behavior

Most drinks are served by opening the bottle and pouring the content in a glass, for that, Drink gets the ShowServingInstructions() method. Wine however, requires a special glass, should be opened in front the client and someone should taste it first. This is a whole new implementation of the serving.

- We need a **common behavior** for Drinks
- We need to be **able (not required!) to replace** that common behavior
- Solution: we make the method **virtual** in the base class and override it only if necessary

```
public abstract class Drink
{
    public virtual void ShowServingInstructions()
    {
        //serving instructions for most drinks (eg. soda)
    }
```

```
public class Wine : Drink, ICollectable
{
    //virtual method in baseclass => CAN be overriden
    //      (in this case: replace all default behavior)
    public override void ShowServingInstructions()
    {
        //show specific serving instructions for wine
    }
```

This 'virtual' keyword can be used for both methods (as in the example) and properties!

Adding behavior

There are two ways to add behavior (override virtual + base, override abstract).

- We want to add behavior for **specific types**
  ➔ using a subclass
- We have common behavior in the base class that we wish to **extend** with extra behavior?
  ➔ virtual
- We don't have common behavior in the base class but want to **force** the subclasses **to implement** specific behavior? ➔ Interface or abstract

Using a subclass

Situation: We want to add behavior for **specific types**.

- How: add properties and methods in the subclass
- Example: Only Wine contains data for the Country of origin, a list of grapes, a different price for the whole bottle instead of just a glass, and a wine type (red, white,…).
- Solution: we add properties in the Wine class that are not in the Drink base class.

```
public class Wine : Drink, ICollectable   //aut
{
    //extra variables
    public string Country { get; set; }
    public List<string> Grapes { get; set; }
    public double PricePerBottle { get; set; }
    public WineType Type { get; set; }
```

Virtual

Situation: We have common behavior in the base class that we wish to **extend** with extra behavior.

- How: the base class contains an implementation for the behavior (property/method). The subclass extends this by overriding the behavior but also **calls the base** behavior
- Example: Most drinks are to be shown by the name of the product (eg.: coca-cola). The name of the Wines, however, should be preceded by the type ("RED", "WHITE",…)
- Solution: we override the property, call the base class and add extra behavior.

```
public abstract class Drink
{
    public virtual string Name { get; set; }
```

```
public class Wine : Drink, ICollectable   //automatically in
{
    public override string Name
    {
        get { return Type.ToString() + " - " + base.Name; }
        set { base.Name = value; }
    }
```

Interface / abstract

Situation: We don't have common behavior in the base class but want to **force** the subclasses **to implement** specific behavior.

- Abstract - how: the base class contains an **abstract** function header or property, which by consequence has no implementation. All subclasses **must** implement this function or they get an error. Abstract properties and methods can exist in a default or abstract class, together with non-abstract ones.
- Interface - how: an interface is a type of class that contains **no implementation** for any properties or methods. Every class that implements (not: inherits) this interface **must** provide an implementation for all properties and methods in this interface, or they get an error.
- Interfaces and abstract CLASSES make sure that the user cannot instantiate the base class. For example, if I have an IDrink interface, or an abstract class Drink, the user will **not** be able to do as follows:

  ```
  Drink drink = new Drink();
  ```

  Which is good, since 'Drink' itself does not exist in this case. However, polymorphism will allow them to do this:

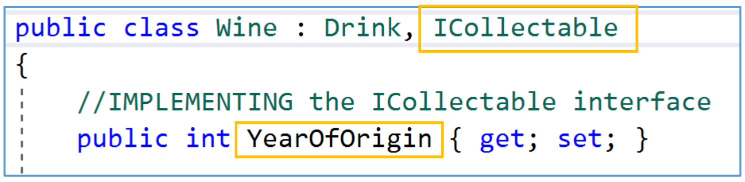
  ```
  Drink drink = new Soda();
  ```

  For more information on this, we refer to the explanation in the theory lesson.
- Example - abstract: every specific type of drink gets a 'sign' (could be an icon, or in this case: B for Beer, W for Wine, …). Since 'Drink' does not exist, we have no specific sign for it.
- Solution - abstract: we create an abstract property 'Sign' with only a getter, forcing all subclasses to provide an implementation for this property. In this way, using polymorphism, we can display every type of drink with its sign.
  Also, we make the Drink class itself abstract so that it cannot be instantiated.

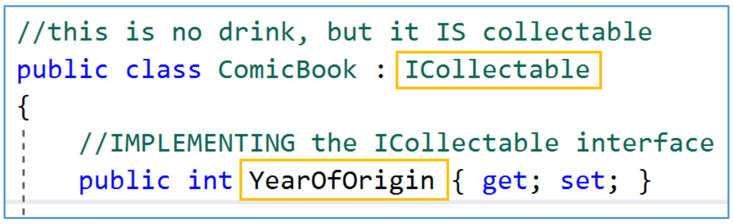
  ```
  public abstract class Drink
  {
      //abstract variables: MUST be overriden by subclasses
      public abstract string Sign { get; }
      public abstract string Description { get; }
  ```

  ```
  public class Wine : Drink, ICollectable
  {
      public override string Sign
      {
          get { return "W"; }
      }
  ```
- Example - interface: we wish to use some of the drinks in another app that visualizes all kinds of collectable items (wines, comic books, etc.). What is important for that app is the year of origin.
- Solution - interface: we make an interface that can be implemented by all kinds of collectable items.

  ```
  public interface ICollectable
  {
      int YearOfOrigin { get; set; }
  }
  ```



```
public class Wine : Drink, ICollectable
{
    //IMPLEMENTING the ICollectable interface
    public int YearOfOrigin { get; set; }
```



```
//this is no drink, but it IS collectable
public class ComicBook : ICollectable
{
    //IMPLEMENTING the ICollectable interface
    public int YearOfOrigin { get; set; }
```

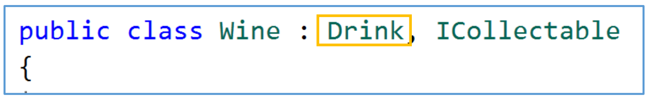
Inheritance vs Object composition

Below you can find a few examples from the demo. For further explanation on this topic, we refer to the theory lessons.

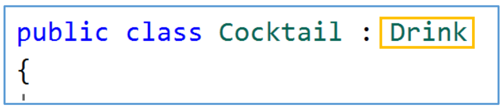
Inheritance

Inheritance defines a **… is a …** relationship.

- Example: a Cocktail **is a** Drink, a Wine **is a** Drink, …



```
public class Wine : Drink, ICollectable
{
```

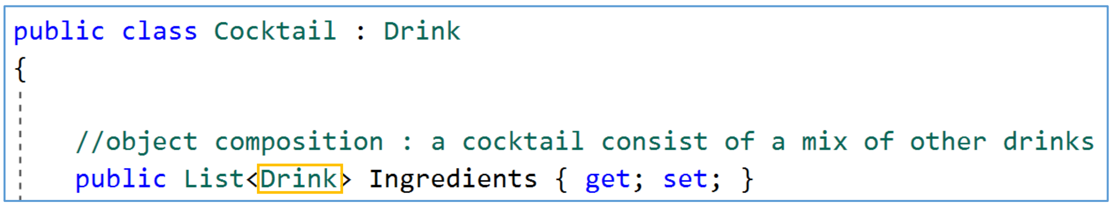


```
public class Cocktail : Drink
{
```

Object composition

Object composition defines a **… has a …** relationship, resulting in a **property** of that type.

- Example: a Cocktail **has a** list of ingredients (of type Drink)



```
public class Cocktail : Drink
{

    //object composition : a cocktail consist of a mix of other drinks
    public List<Drink> Ingredients { get; set; }
```