# Working with Class Hierarchies

**Gill Cleeren**
CTO Xpirit Belgium

@gillcleeren | xpirit.com/gill

# Overview

Adding inheritance

Working with polymorphism

Exploring sealed and abstract classes

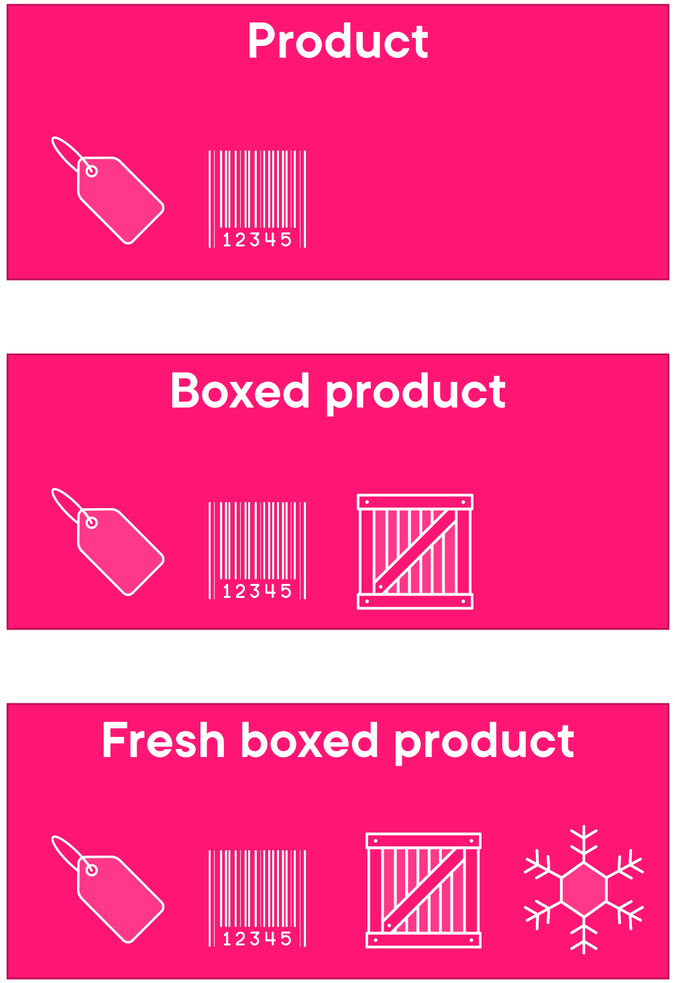
Using extension methods

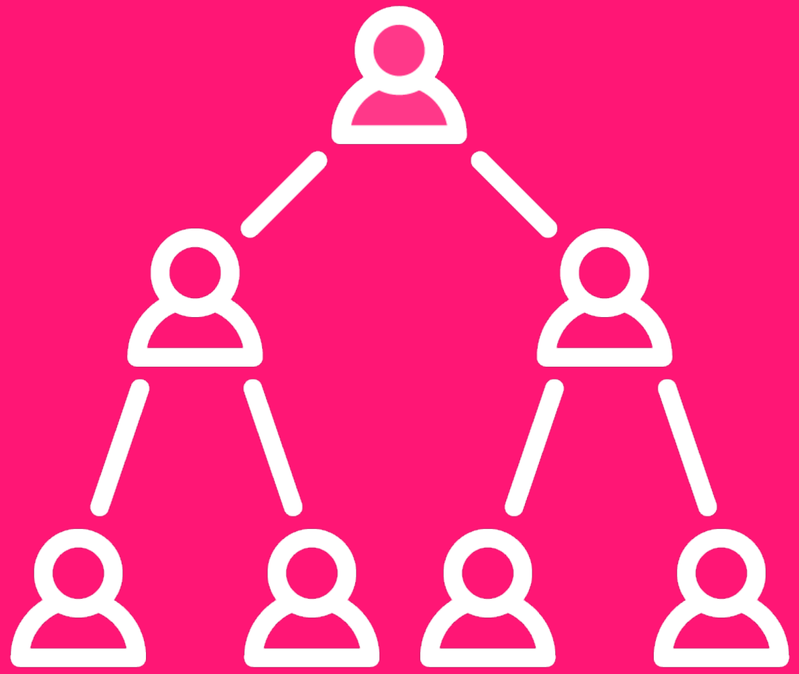
# Adding Inheritance

“We have in fact different types of products. They behave rather similar but based on the type, they will be differences.

Can we build this into the application?”

# Different Types of Products





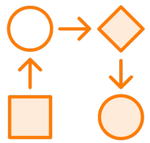
# What are we doing here?

Organizing classes

Makes it easier to understand what they will ‘inherit’ from another class

This is inheritance, an important OOP pillar

# Introducing Inheritance



Relation between classes

Allow reuse of functionalities of base (parent) class for derived (child) class

Creates the “Is-A” relation

Access to members is controlled using access modifiers

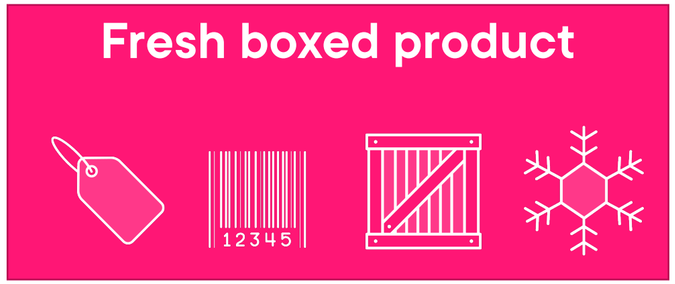
Improves maintainability by avoiding code duplication

# Inheritance Terminology


Parent class
Base class
Superclass


Child class
Derived or extended class
Subclass

Can be multiple levels deep
Class can only have one direct parent



```
public class BaseClass
{
}

public class DerivedClass: BaseClass
{
}
```

# Base and Derived Classes

**Notice the :**

# Creating the Base and Derived Classes

**Product.cs**

```
public class Product
{
    public void UseProduct(int items)
    {
    }
}
```

**BoxedProduct.cs**

```
public class BoxedProduct : Product
{

}
```

```
BoxedProduct newProduct = new BoxedProduct();
newProduct.UseProduct(10);
```

# Using the BoxedProduct

Has access to members of the base Product class
Depends on access modifier though

# The Access Modifiers Revisited

public

private

protected

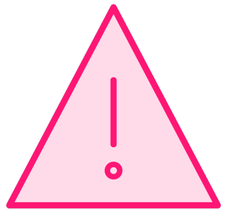
# Accessing the Base Class Members

```
public class Product
{
    public string name;

    public void DisplayDetails()
    {

    }
}
```

```
public class BoxedProduct: Product
{
    public void
        DisplayBoxedProductDetails()
    {
        Console.WriteLine(name);
    }
}
```

# Working with private

```
public class Product
{
    private string name;

    public void DisplayDetails()
    {

    }
}
```



```
public class BoxedProduct: Product
{
    public void
        DisplayBoxedProductDetails()
    {
        Console.WriteLine(name);//error
    }
}
```

# Using the protected Access Modifier

```
public class Product
{
    protected string name;

    public void DisplayDetails()
    {

    }
}
```

```
public class BoxedProduct: Product
{
    public void
        DisplayBoxedProductDetails()
    {
        Console.WriteLine(name);
    }
}
```

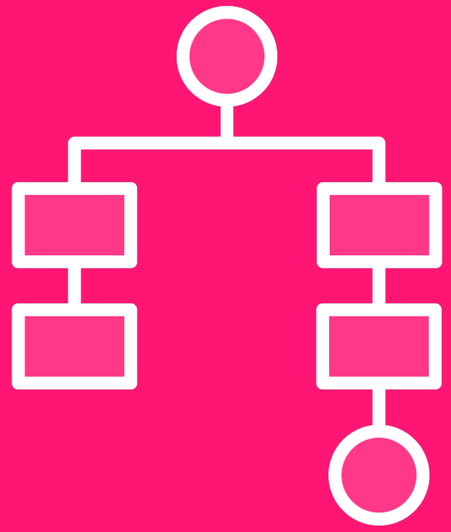
# Extending the Derived Class

**Product.cs**

```
public class Product
{

    public void UseProduct(int items)
    {
    }
}
```

**BoxedProduct.cs**

```
public class BoxedProduct : Product
{
    private int AmountInBox;

    public void UseBoxedProduct(int
        items)
    {
    }
}
```



# Inheritance Is Transitive

Child class will inherit from all ancestors
(multiple levels)

“I think we’ll have products that we store per item, some come boxed, some are bulk products and yes, we also have fresh products which expire quickly.”

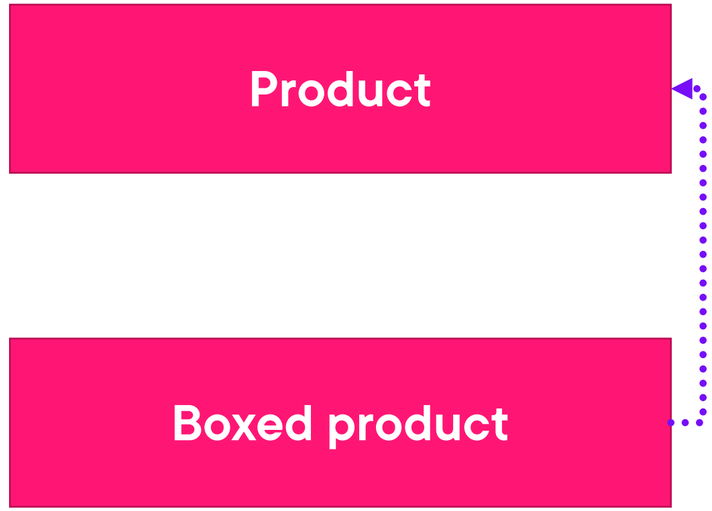
# Demo

Adding inheritance

Creating the different product classes

# Inheritance and Constructors

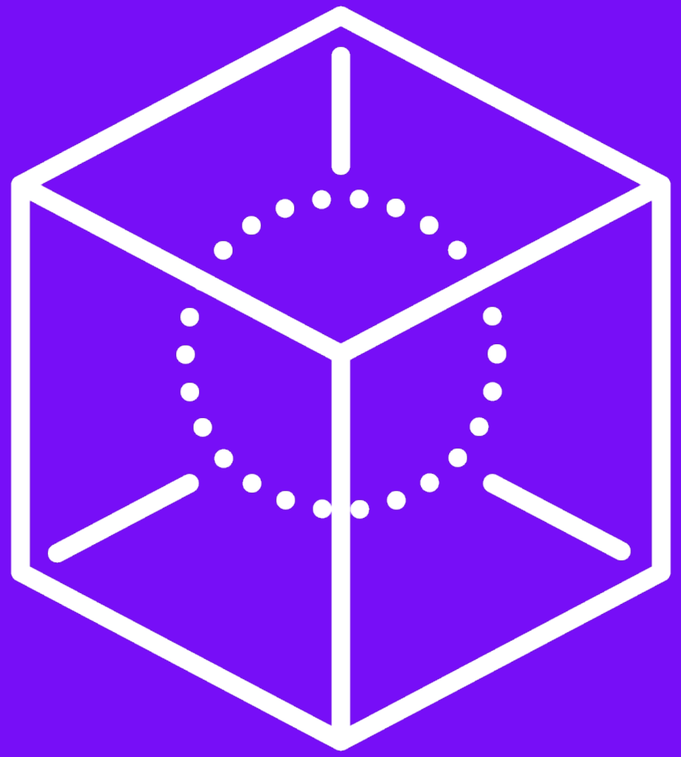
```
BoxedProduct bp = new BoxedProduct();
```



```
public class BoxedProduct : Product
    {

        public BoxedProduct(int id, string name, string? description, Price price, int
             maxAmountInStock, int amountPerBox) : base(id, name, description, price,
             UnitType.PerBox, maxAmountInStock)
        {
            AmountPerBox = amountPerBox;
        }
}
```

## Calling the Base Constructor with Parameters

Base constructor needs to get values for its constructor and will execute before the derived class constructor executes
Uses the base keyword, passing in list of arguments



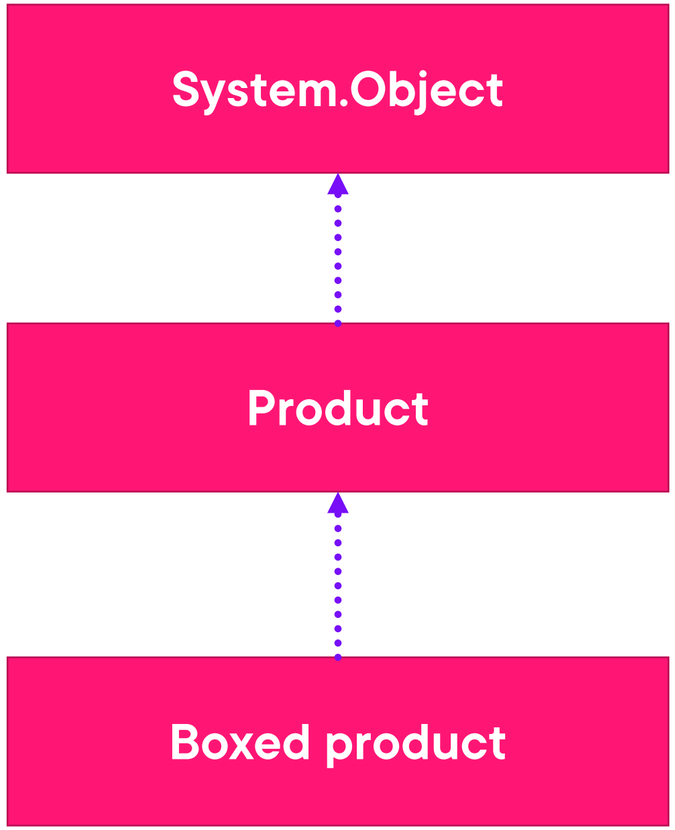
# Everything IS AN object

All types derive from System.Object

Has a parameterless constructor

All types inherit a few members like ToString()

# Inheriting from System.Object



# Demo

## Inheriting from System.Object

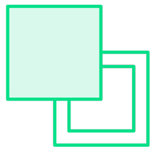
# Working with Polymorphism

# We may have a problem...

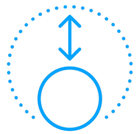
What if the behavior is different?

Derived may want to change an existing base implementation

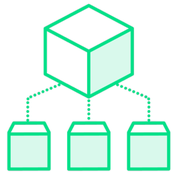
# Introducing Polymorphism



Poly and morph > “many-shaped”



Allow treating objects of derived class as base class object



Based on virtual and override keywords

Override base class method with new implementation

Parameter lists need to be the same, otherwise overloading

# The “Is-A” Relation

```
Product p1 = new BoxedProduct();
Product p2 = new FreshProduct();
p1.UseProduct(10);
p2.UseProduct(8);
```

## Using a Base Reference

We invoke the UseProduct() on the base Product class

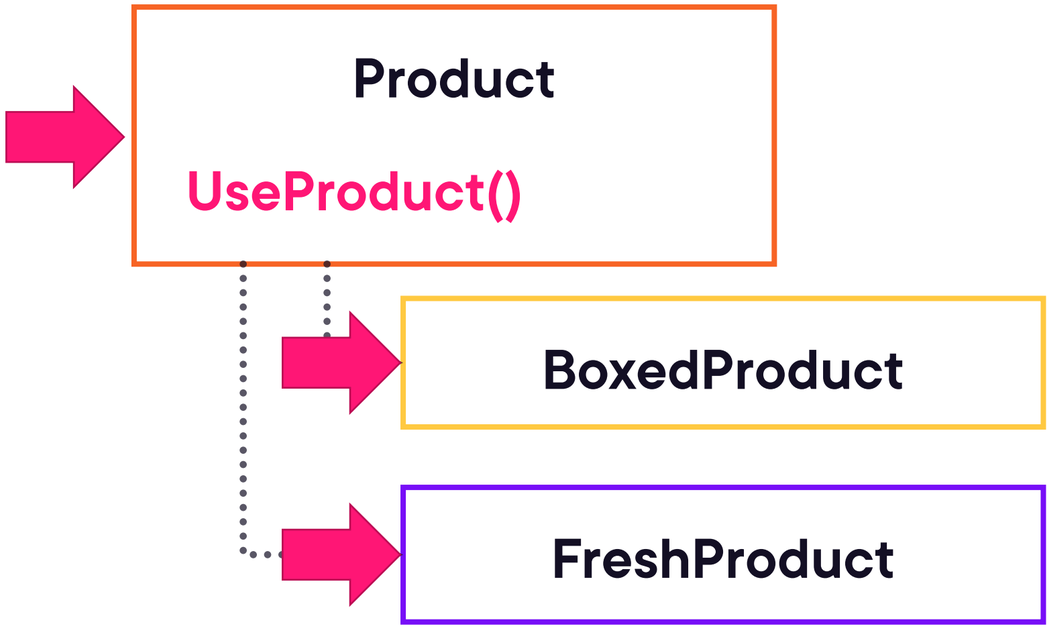
```
BoxedProduct boxedProduct = new BoxedProduct();

SaveProduct(boxedProduct);


public void SaveProduct(Product p)
{

}
```

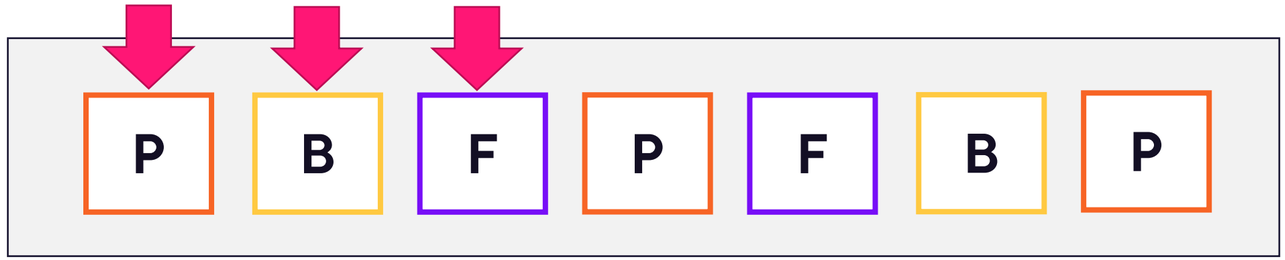
# Using Implicit Conversions

```
BoxedProduct boxedProduct = new BoxedProduct();
Product product = new Product();

Product[] products = new Product[10];

products[0] = product;
products[1] = boxedProduct;
```

## Creating an Array of Products

# Using the Base Reference



# Adding Polymorphism

## Using virtual and override

**Product**

```
public class Product
{
     public virtual void UseProduct
          (int items)
     {
          //base implementation
     }
}
```

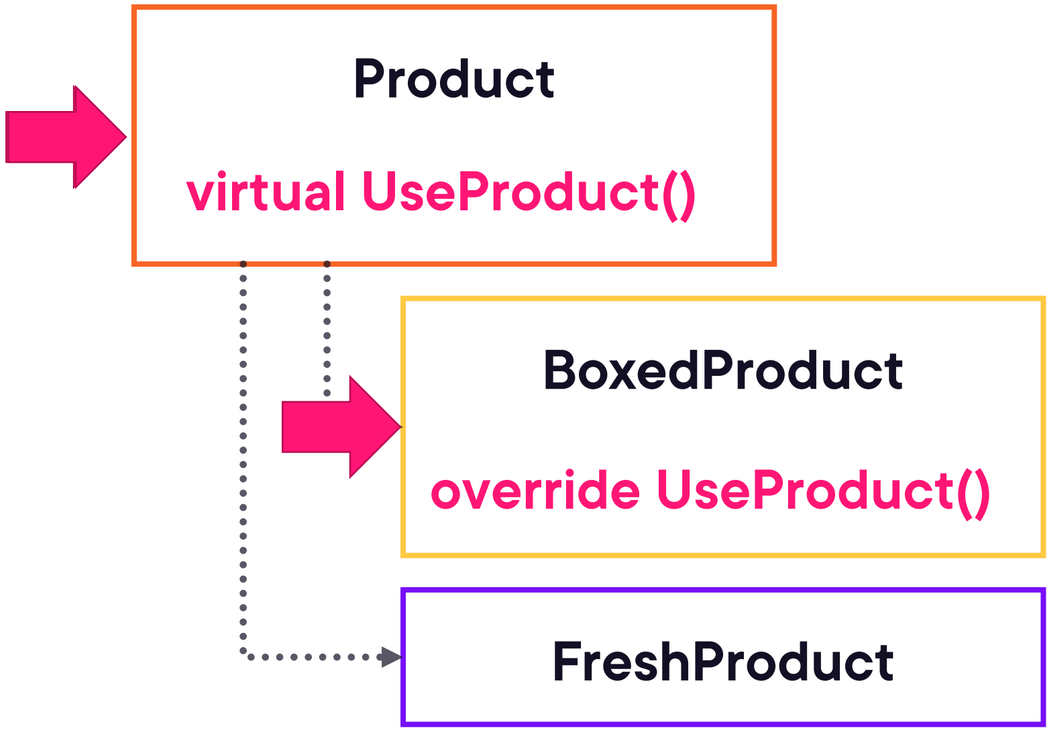
**BoxedProduct**

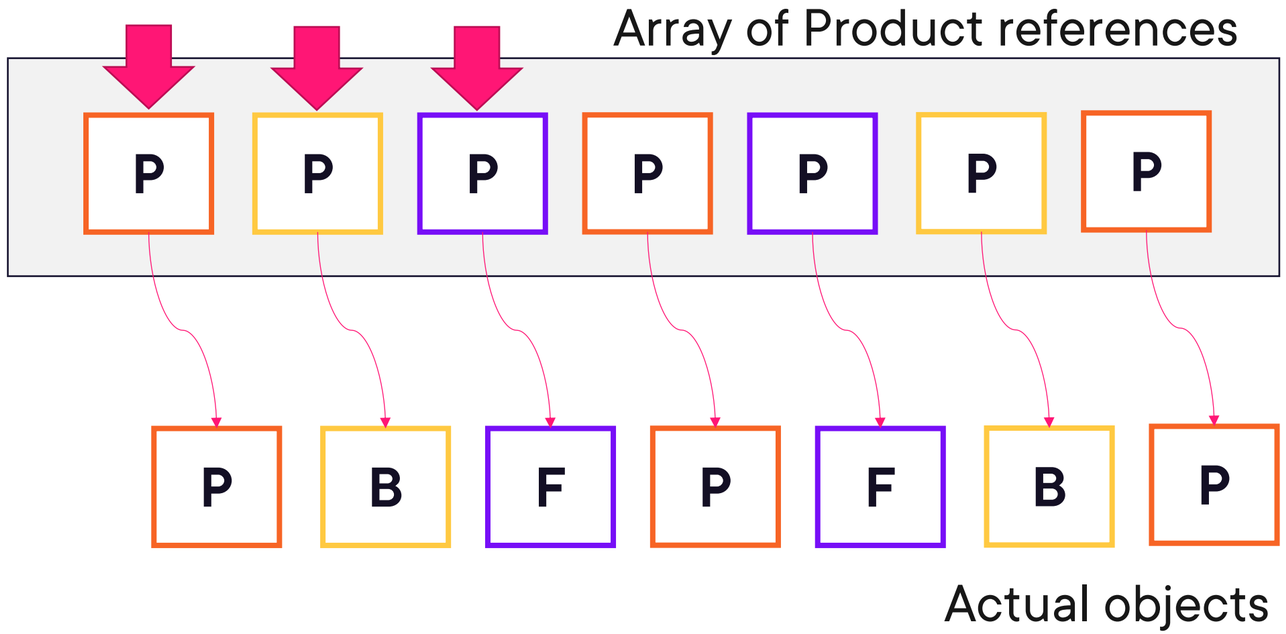
```
public class BoxedProduct: Product
{

     public override void UseProduct
          (int items)
     {
          //custom implementation
     }
}
```

# How does C# know which implementation to use?

The most specific implementation will be selected based on the type of the object!

# Looping over an Array of Product References



# Demo

## Working with polymorphism

```
Product product = new BoxedProduct();

product.UseBoxedProduct();//error
```

# Unavailable Methods

```
public void SaveProduct(Product p)
{
     BoxedProduct bp = (BoxedProduct)p;//explicit cast which can be dangerous

     if (p is BoxedProduct)
     {

     }
}
```

## Using the is Keyword

```
public static void SaveProduct(Product p)
{
    BoxedProduct? bp = p as BoxedProduct;

    if (bp != null)
    {
    }
}
```

# Using the as Keyword

```
public static void SaveProduct(object o)
{
    string p = o.ToString();
}
```

## Working with object

# Explicitly calling the Base Implementation

```
public override void UseProduct(int items)
{
    int smallestMultiple = 0;
    int batchSize;

    while (true)
    {
        smallestMultiple++;
        if (smallestMultiple * AmountPerBox > items)
        {
            batchSize = smallestMultiple * AmountPerBox;
            break;
        }
    }

    base.UseProduct(batchSize);

}
```

```
public class Product
{
     public virtual void UseProduct(int items)
     {
          //base implementation
     }
}
public class BoxedProduct: Product
{
     public void UseProduct(int items)
     {
          //custom implementation
     }
}
```

# Omitting the override Keyword

Hides the base implementation, known as shadowing
Triggers compiler warning

```
public class Product
{
     public virtual void UseProduct(int items)
     {
          //base implementation
     }
}
public class BoxedProduct: Product
{
     public new void UseProduct(int items)
     {
          //custom implementation
     }
}
```

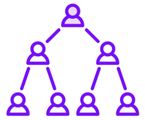
# Adding the new Keyword

Clears the compiler warning
Explicit about the intention

# Exploring Abstract and Sealed classes

“I’m wondering… Should it still be possible to create a Product instance?”

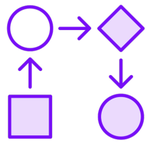
# Introducing Abstract Classes



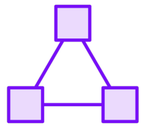
Can’t be instantiated but can be inherited

Contains shared functionality for its child classes

Can define abstract methods, without implementation



Child classes can be instantiated (if not declared abstract too)



Based on abstract keyword

# Creating and Using an Abstract Class

## Instantiating won’t work

**Product**

```
public abstract class Product
{
    public virtual void UseProduct
        (int items)
    {
        //base implementation
    }
}
```

**Program.cs**

```
Product p1 = new Product();//error
```

```
Product p1 = new BoxedProduct();
```

## This Will Work!

Base reference points to inheriting class

```
public abstract class Product
{
    public abstract void UseProduct(int items);
}
```

## Adding abstract Methods

Don’t have an implementation
Must be overridden in inheriting types (otherwise, inheritor becomes abstract too)
virtual methods don’t need to receive an override in the inheriting type

# Demo

## Making the Product class abstract

```
public sealed class Product
{}

public class BoxedProduct: Product // error
{}
```

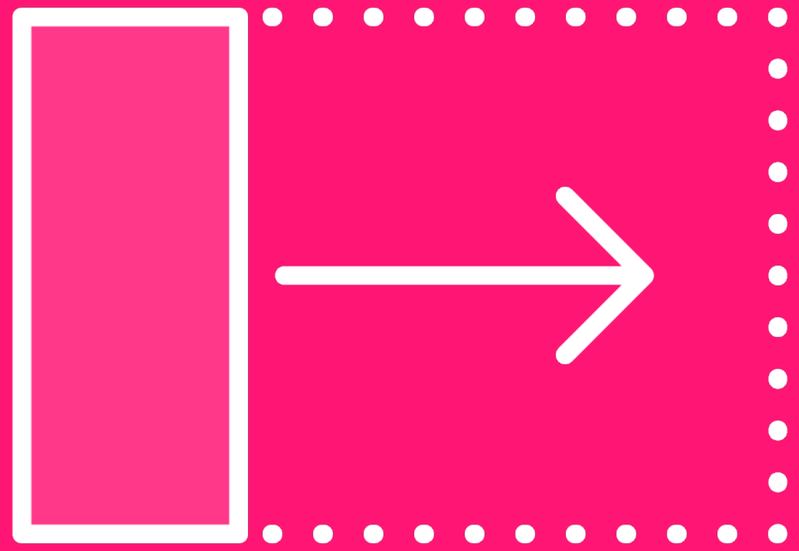
# Making a Class sealed

Inheriting isn’t allowed
Useful if creating libraries that you want to protect

# Demo

Making use of sealed classes

# Using Extension Methods



# We may want to add extra functionality to a class

Inherit from existing class

Change the original

But what if we don’t have access? Or it’s sealed?

Create an extension method

```
public sealed class Product
{
}

public static class ProductExtensions
{

    public static double RemoveProduct(this Product product)
    {
    }
}
```

## Adding an Extension Method

Allow to “add” methods on any type, even part of .NET or written by someone else
Must be static inside a static class and requires use of this
Type is the type that will be extended

```
Product p1 = new Product();

p1.RemoveProduct();
```

# Using an Extension Method

# Demo

## Adding and using an extension method

# Summary

Inheritance allows us to move member to the base type

Inheriting types can extend the base class and use its functionality and data

Access modifiers control what can be accessed by inheriting classes

Polymorphism allows to create different version of method in inheriting types

Using extension methods, all types can be extended with extra functionality

Up Next:

# Reusing Code through Interfaces