

## Chapter 14

# How to work with inheritance

# Objectives

## Applied

1. Use any of the features of inheritance that are presented in this chapter as you develop the classes for your applications.

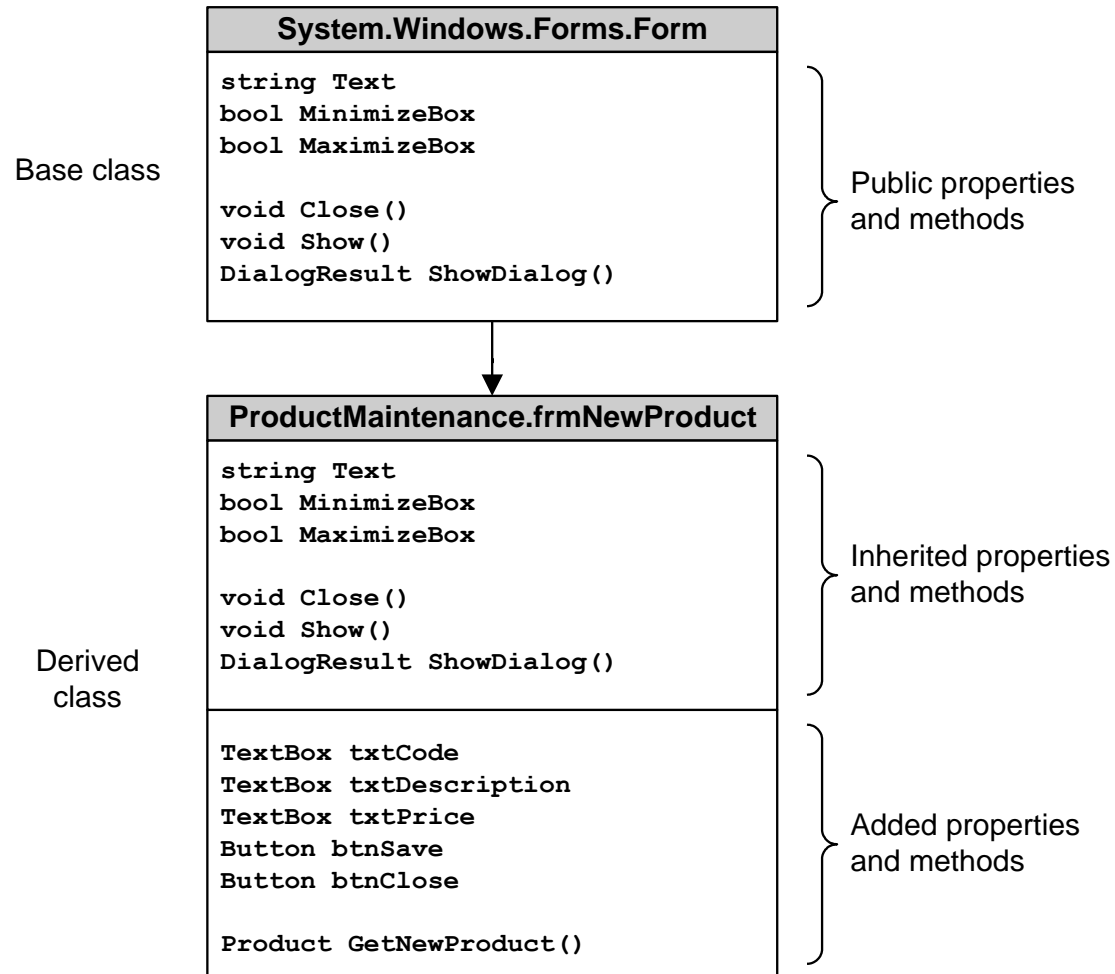
## Knowledge

1. Explain how inheritance is used within the .NET Framework classes.
2. Explain why you might want to override the ToString, Equals, and GetHashCode methods of the Object class as you develop the code for a new class.
3. Describe the use of the protected and virtual access modifiers for the members of a class.
4. Describe the use of polymorphism.
5. Distinguish between an abstract class and a sealed class.

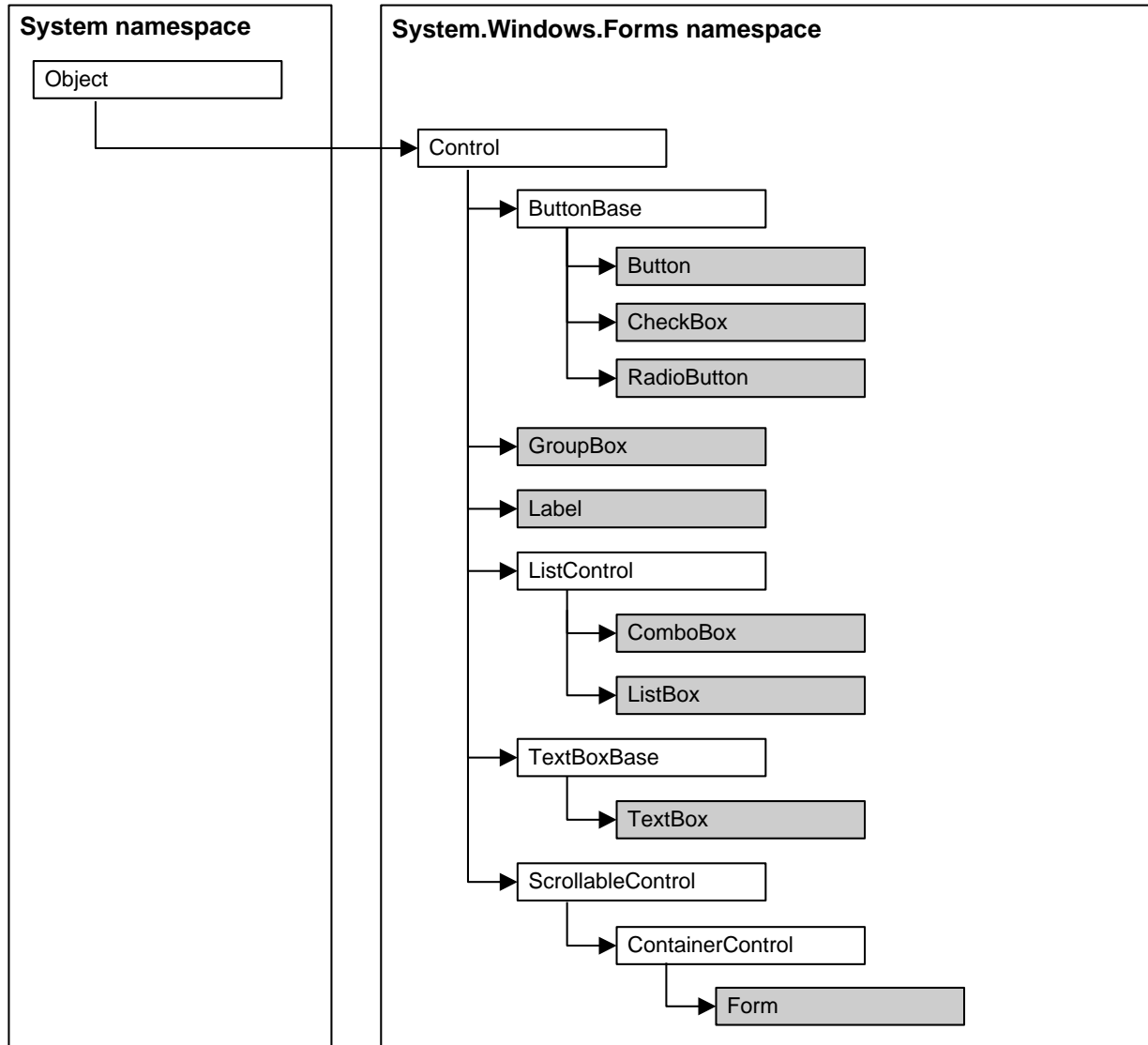
# Inheritance

- Inheritance is the object oriented programming concept that allows you to create a base class and then derive other classes from that class.
  - Think about inheritance in a genetic context. A parent has all kinds of behaviors and characteristics. Children inherit lots of those behaviors and characteristics. They add additional behaviors and characteristics and learn to do some of the existing behaviors differently.
  - Inheritance in a programming context is just like that.
  - You use inheritance every time you create a form
    - The form class “knows how to be a form”
    - Your form is derived from the form class so it knows all of those thing too. Then you add the functionality that’s specific to your form.

# How inheritance works



# The inheritance hierarchy for form control classes



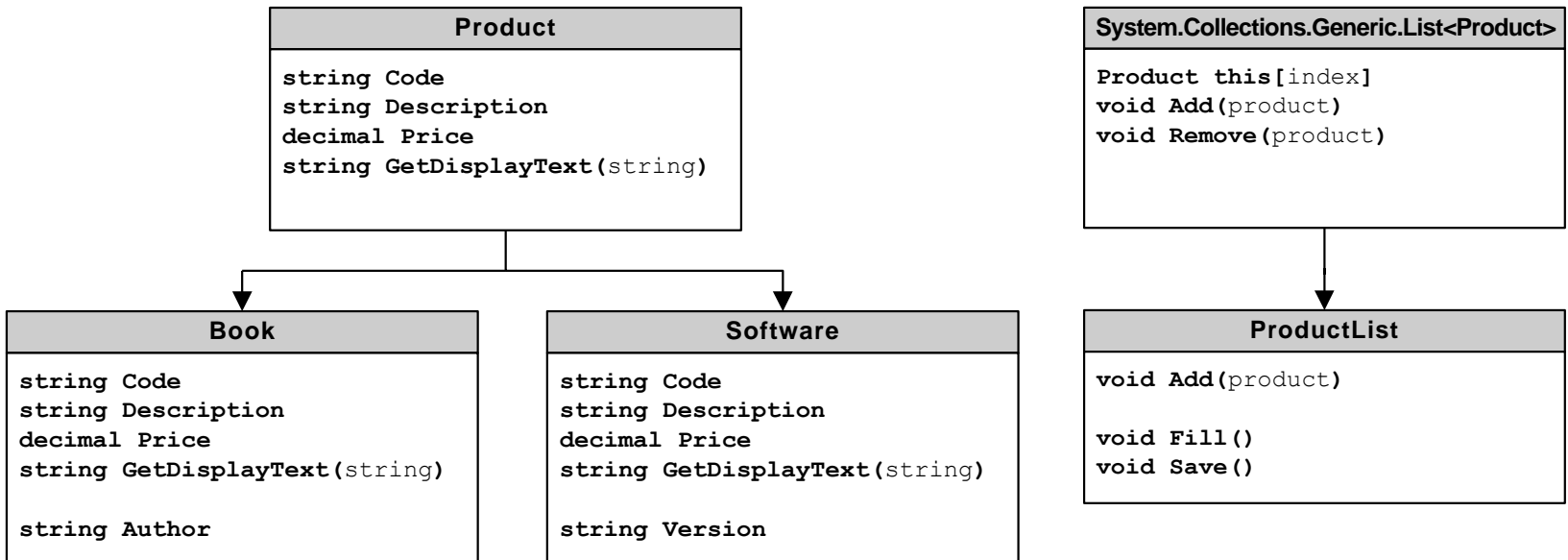
# The Object Class

- Is the base class for every class in the .net framework and for every class that you create.
  - You've seen the object class before ... the first parameter for an event handler is the sender which is some control on the form ... it might be a textbox or a button or a menu so the datatype used for the parameter is the most generic datatype possible ... an object.
  - You override ToString which is defined in the Object class. The behavior for an object is to return the name of the class ... which is not particularly helpful ... so you create a more specific version for your class.
  - We talked about overriding Equals (and GetHashCode) last time. Again, the default behavior isn't very useful because it can only check to see if 2 objects refer to the same object. If you want any Ace of Clubs to be equal to any other Ace of Clubs you could override Equals.

# Methods of the System.Object class

- **ToString()**
- **Equals(object)**
- **Equals(object1, object2)**
- **ReferenceEquals(object1, object2)**
- **GetType()**
- **GetHashCode()**
- **Finalize()**
- **MemberwiseClone()**

# Business classes for a Product Maintenance application





# The code for the Product class

```
public class Product
{
    private string code;
    private string description;
    private decimal price;

    public Product()
    {
    }

    public Product(string code, string description,
        decimal price)
    {
        this.Code = code;
        this.Description = description;
        this.Price = price;
    }
}
```

## The code for the Product class (cont.)

```
public string Code
{
    get
    {
        return code;
    }
    set
    {
        code = value;
    }
}

public string Description
{
    get
    {
        return description;
    }
}
```

## The code for the Product class (cont.)

```
        set
        {
            description = value;
        }
    }

    public decimal Price
    {
        get
        {
            return price;
        }
        set
        {
            price = value;
        }
    }
}
```

## The code for the Product class (cont.)

```
public virtual string GetDisplayText(string sep)
{
    return Code + sep + Price.ToString("c") + sep
           + Description;
}
}
```

# The code for the Book class

```
public class Book : Product
{

    public Book()
    {
    }

    public Book(string code, string description,
                string author, decimal price) : base(code,
                description, price)
    {
        this.Author = author;
    }

    public string Author
    {
        get
        {
            return author;
        }
    }
}
```

## The code for the Book class (cont.)

```
        set
        {
            author = value;
        }
    }

    public override string GetDisplayText(string sep)
    {
        return base.GetDisplayText(sep) + " ("
            + Author + ") ";
    }
}
```

# The code for the Software class

```
public class Software : Product
{

    public Software()
    {
    }

    public Software(string code, string description,
        string version, decimal price) : base(code,
        description, price)
    {
        this.Version = version;
    }

    public string Version
    {
        get
        {
            return version;
        }
    }
}
```


## The code for the Software class (cont.)

```
        set
        {
            version = value;
        }
    }

    public override string GetDisplayText(string sep)
    {
        return base.GetDisplayText(sep) + ", Version "
            + Version;
    }
}
```



## Code that uses the overridden methods

```
Book b = new Book("CS10", "Murach's C# 2010",  
    "Joel Murach", 54.50m);  
Software s = new Software("NPTK",  
    ".NET Programmer's Toolkit", "4.0", 179.99m);  
Product p;   
p = b;  
MessageBox.Show(p.GetDisplayText("\n"));  
                // Calls Book.GetDisplayText  
  
p = s;  
MessageBox.Show(p.GetDisplayText("\n"));  
                // Calls Software.GetDisplayText
```

**THIS IS AN EXAMPLE OF POLYMORPHISM, which means many shapes. The “right” version of the method GetDisplayText is called based on the type of object p refers to.**

# Your Turn

- The first part of lab 1 asks you to create a set of Customer classes: WholesaleCustomer and RetailCustomer (p 475)
- Before you begin, find a copy of your Customer Maintenance solution from labs 4, 5 and 6 from 233N. (Let me know if you don't have a copy and I'll share mine.) You can either make a copy of the ENTIRE SOLUTION or use the one from last term.
  - The Customer class should change in 2 ways
    - Make each instance variable protected
  - Add the 2 new classes (to your class library project) as described in the exercise at the end of chapter 14.
  - Test each of the new classes in your console app (since you already have that in your solution) or add a unit test project and test your new classes there.

# Relationships Between Classes

- Inheritance is an “is a” relationship.
  - A WholesaleCustomer “is a” customer.
- When you created the ProductList class in the last lab
  - It has a List<Product> as it’s instance variable
  - This is a “has a” relationship.
  - A Deck “has a” list. A Hand “has a” list.
- It’s also possible to create a ProductList that “is a” list. Instead of having List as an instance variable, it will be derived from a List.

## The code for the ProductList class

```
public class ProductList : List<Product>
{
    // Modify the behavior of the Add method of the
    // List<Product> class
    public new void Add(Product p)
    {
        base.Insert(0, p);
    }

    // Provide two additional methods
    public void Fill()
    {
        List<Product> products = ProductDB.GetProducts();
        foreach (Product product in products)
            base.Add(product);
    }
    public void Save()
    {
        ProductDB.SaveProducts(this);
    }
}
```

# Your Turn

- The second part of lab 1 asks you to create a new version of the CustomerList class that IS A List<Customer> instead of HAS A List<Customer> as an instance variable. (p 476)
  - Add a new class to your class library. Call it CustomerList2 and
    - Derive it from List<Customer>
    - Make sure it has NO INSTANCE VARIABLES
    - Write just the methods that are described in the exercise at the end of the chapter.
    - Test it as you did your original CustomerList but change the variable definitions in your tests to CustomerList2. Which tests run as is? Which things don't compile. What would you have to do to make those tests work? As long as the methods that the text talks about work, there's no need to fix all of your tests. Just comment out the things that you did as "extra credit" last term.

# What's Next?

- The third part of lab 1 asks you to add classes to your set of Card related classes.
  - BlackjackHand derived from the Hand class.
    - It will NOT have an instance variable that's a List<Card>. WHY?
    - It will NOT need methods like Add, HasCard, Discard. WHY?
    - What additional methods do BlackjackHands need that Hands don't have already?
      - Score
      - IsBusted
      - HasAce is a method that I'll have you write in order to make writing Score easier.
    - Let's write an algorithm for calculating the score.
  - Test it using either a console app or NUnit tests.

# What's Next?

- You'll notice that I'm not asking you to do the GUI for the CustomerMaintenance app.
- You can also begin to designing a GUI for playing blackjack. (This is lab 3)
  - Add a new Windows Forms App to your Card solution from last term.
  - Steal UI stuff from Concentration.
    - Copy picture box controls. Be careful when you name the picture boxes for the player and the dealer. You might not want to call them card1, card2, card3 etc.
    - Identify the instance variables that you'll need. A deck, 2 BJHands.
    - Copy methods that you think you might need. You'll probably need to change some things in each method but we'll talk about all of that during the last week of classes.

# What's Next

- Chapter 15 and Reading Quiz 3
- Lab 2 – Part is out of the book and part adds classes to MTDominos
- Lab 3 – GUI to play BlackJack
- Quiz 1
- Database chapters