Chapter 15

Abstract Classes and Interfaces

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.

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 term. 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.

Relationships Between Classes

- Inheritance is an “is a” relationship.
  - A WholesaleCustomer “is a” customer.
  - A BlackJack Hand “is a” Hand.
  - A PrivateTrain “is a” Train
- When you created the ProductList class in the last lab
  - It has a List<Product> as it’s intance variable
  - This is a “has a” relationship.
  - A Deck “has a” list. A Hand “has a” list.
  - A BoneYard “has a” list. A Train “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.

Abstract Classes

- The end of chapter 14 talks about some things that you should know about but aren’t in this lab.
  - An abstract class is a class that is useful as part of the design of a hierarchy of classes but that is not fully implemented and therefore cannot be instantiated.
    - You created a product, a book and a software class. We don’t really sell products, we sell books and software so it might have been cleaner in terms of design to make the product an abstract class. In the product class you’d do as much of the implementation as you can and identify the things that derived classes must add to finish the implementation.
    - In some ways, it’s a “contract” between class developers that specifies how a set of classes must work so polymorphism works. You want to be able to create an array of products and fill it with software and books and have both classes have some methods in common, like GetDisplayText.

Abstract Classes

- The syntax for creating an abstract class essentially involves adding the keyword abstract to the heading of the class definition.
- There’s an example on page 470-471 of your text. That example is in the next couple of slides

An abstract Product class

```
public abstract class Product
{
    public string Code {get; set; }
    public string Description {get; set; }
    public decimal Price {get; set; }

    public abstract string GetDisplayText(string sep);
    // No method body is coded.
}
```

An abstract read-only property

```
public abstract bool IsValid
{
    get;            // No body is coded for the get accessor.
}
```

A class that inherits the abstract Product class

```
public class Book : Product
{
    public string Author {get; set; }

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

Sealed Classes

- A sealed class is a class from which you can not derive another class.
  - In practice, I have never used one of these.

Interfaces

- Chapter 15 introduces the concept of an interface
- Like Abstract classes, an interface is a contract between programmers that specifies the properties and methods that a class that implements an interface must write.
  - In order to be able to sort a list, the objects in the list must be able to be compared … they must implement the IComparable interface … sorting relies on it and sorting will break if the class doesn't have the CompareTo method that sorting requires
  - In order to be able to iterate through a class that contains a list as an instance variable using a for each loop the class must implement the IEnumerable interface … the foreach loop relies on a method called GetEnumerator and the foreach loop won't work if that method isn't defined in the list class (like a hand)
- Interfaces, unlike abstract classes, identify method and property signatures but have NO implementation

Interfaces

- C# allows a class to be derived from one class only
  - Most languages allow only single inheritance but some languages like C++ allow multiple inheritance
  - An SUV for example, could be derived from both a car and a truck in C++ but not in C#
- C# allows a class to implement multiple interfaces
  - A book could be derived from a product and could implement IDisplayable and IComparable and IClonable (so that any code that uses it can be assured that it can be printed and copied and compared)
- The next couple of slides show a simple example from page 479 in the text.

The IDisplayable interface

```
interface IDisplayable
{
    string GetDisplayText(string sep);
}
```

The code for the cloneable Product class

```
public class Product : IDisplayable,
                       ICloneable, IComparable<Product>
{
    public string Code { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }

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

    #region IDisplayable Members
```

The code for the cloneable Product class (cont.)

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

#endregion

#region ICloneable Members

public object Clone()
{
    Product p = new Product();
    p.Code = this.Code;
    p.Description = this.Description;
    p.Price = this.Price;
    return p;
}
#endregion
```

A class that implements the IComparable interface

```
        #region IComparable<Product> Members

        public int CompareTo(Product other)
        {
            return this.Code.CompareTo(other.Code);
        }

        #endregion
    }
```

Code that creates and clones a Product object

```
Product p1 = new Product("JAV5",
    "Murach's Beginning Java 2, SDK 5", 52.50m);
Product p2 = (Product)p1.Clone();
p2.Code = "JSE6";
p2.Description = "Murach's Java SE 6";
Console.WriteLine(p1.GetDisplayText("\n") + "\n");
Console.WriteLine(p2.GetDisplayText("\n") + "\n");
```

The output that's displayed by this code

```
JAV5
Murach's Beginning Java 2, SDK 5
$52.50

JSE6
Murach's Java SE 6;
$52.50
```

Code that uses the IDisplayable interface

```
IDisplayable product = new Product("CS10",
    "Murach's C# 2010", 54.5m);
Console.WriteLine(product.GetDisplayText("\n"));
```

Code that uses the Product class

```
Product p1 = new Product("VB10",
   "Murach's Visual Basic 2010", 54.50m);
Product p2 = new Product("CS10", "Murach's C# 2010",
   54.50m);
int compareValue = p1.CompareTo(p2);
if (compareValue > 0)
    Console.WriteLine("p1 is greater than p2");
else if (compareValue < 0)
    Console.WriteLine("p1 is less than p2");
else if (compareValue == 0)
    Console.WriteLine("p1 is equal to p2");
```

.NET Interfaces

- The next couple of slides list some interfaces that the .NET framework classes rely on.
- We’ve talked about IComparable which you should implement if you want sorting to work.
- Another interface that you’ll often implement in custom list classes (like CustomerList and Hand) is IEnumerable. It gives you the ability to use a foreach loop to iterate through a hand, for example.

Commonly used .NET interfaces

Interface	Members
`ICloneable`	`object Clone()`
`IComparable`	`int CompareTo(object)`

`IConvertible`	`TypeCode GetTypeCode()`
	`decimal ToDecimal()`
	`int ToInt32()`
	`(etc.)`

`IDisposeable`	`void Dispose()`

Commonly used .NET interfaces for collections

Interface	Members	
`IEnumerable`	`IEnumerator GetEnumerator()`	

`IEnumerator`	`object Current`
	`bool MoveNext()`
	`void Reset()`	

`ICollection`	`int Count`
	`bool IsSynchronized`
	`object SyncRoot`
	`void CopyTo(array, int)`	

`IList`	`[int]`
	`int Add(object)`
	`void Clear()`
		`void Remove(object)`
		`void RemoveAt(int)`

`IDictionary`	`[int]`
	`ICollection Keys`
	`ICollection Values`
		`int Add(object)`
		`void Remove(object)`
		`void Clear()`

IEnumerable

- You may have noticed that you couldn’t use a foreach loop (in main or in your UI code or even in your unit tests) with your deck class or your hand class but you could use a for loop. That’s because the foreach loop only works for classes that implement the IEnumerable interface.
- Requires that you write a method called GetEnumerator
- The syntax for the implementation of this method in the ProductList class is on the next slide.
  - You may ask yourself, “where do they come up with this stuff?”. And you would be absolutely right about that for this one. Just do it this way. It works.

A class that implements the IEnumerable<> interface

```
public class ProductList : IEnumerable<Product>
{
    private List<Product> list = new List<Product>();

    // other members

    #region IEnumerable<> Members

    public IEnumerator<Product> GetEnumerator()
    {
        foreach (Product p in list)
        {
            yield return p;
        }
    }

    #endregion
```

A class that implements the IEnumerable<> interface (cont.)

```
    #region IEnumerable Members

    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
    {
        throw new Exception(
            "The method or operation is not implemented.");
    }

    #endregion

}
```

Code that uses the new ProductList class

```
Product p1 = new Product("VB10",
    "Murach's Visual Basic 2010", 54.50m);
Product p2 = new Product("CS10", "Murach's C# 2010",
    54.50m);
ProductList list = new ProductList();
list.Add(p1);
list.Add(p2);
foreach (Product p in list)
{
    Console.WriteLine(p.ToString());
}
```

Your Turn

- The first part of lab 2 is out of the book. Exercise 15-1 and 15-2 on pages 505 and 506. It asks you to
  - Implement ICloneable in the Customer class. 15-1 #3. And test it in either a console app or an nUnit unit test.
  - Implement IEnumerable in the CustomerList class (not CustomerList2. WHY?). 15-2 #2. And test it in either a console app or an nUnit test.
  - NOTICE THAT I HAVEN'T ASSIGNED the part of the exercise that works with the GUI.
  - Make your changes to the classes that you have been working on in 233N and lab 1 from this term. If you need starting files, let me know.

Your Turn

- The last part of lab 2 asks you to add classes to your set of MTD classes.
  - Hand represents a hand of MTDominos
  - PrivateTrain represents the train for a MTD player. It's derived from the Train class.
  - Let's look at the list of methods that you should write for the hand class first. We'll ignore the "PLAY" methods that involve the Train class at first.
  - Let's look at the list of methods that you should write for the private train class next.
  - And then the interaction between the two.
  - Implement any interfaces that seem appropriate to you. Would you like to be able to Clone a domino? Compare 2 dominos? Iterate through a hand or a train using a foreach loop?
  - And the need for Events and Delegates.

Delegates and Events

- In lots of applications there's something in the UI that tells you when you've made a change to the document that you're working on so that you have a visual cue that tells you that you need to save the document.
  - Remember the concept of encapsulation? Is it reasonable to expect the UI to keep track of all of the reasons why a document could need to be saved, or should the document be responsible for letting the UI know it needs to be saved?
- Delegates and Events are the mechanism that allows a class to "broadcast" to other classes that are paying attention that something has happened so that the other class can respond appropriately.
  - You use this mechanism all of the time when you write an event handler to respond to the fact that the user clicked a button on a form.

Delegates and Events

- In our ProductList class, we want to be able to “broadcast” an event, that let’s the UI know when the list has changed so that the UI can ask the user if he/she wants to save before closing the form, for example.
  - The event is called Changed and it is declared and “raised” in the ProductList class.
- The class also agrees to “delegate” the handling of the event to other classes and agrees to provide those classes with the list of products.
  - A delegate specifies the signature of a method that can be used to handle an event.
- THIS IS CONFUSING!! in the abstract but it’s not too hard to see how it might be helpful in a specific example.

The code for the ProductList class

```
public class ProductList
{
    private List<Product> products;

    public delegate void ChangeHandler(
        ProductList products);
    public event ChangeHandler Changed;

    public ProductList()
    {
        products = new List<Product>();
    }

    public int Count
    {
        get
        {
            return products.Count;
        }
    }
```

The code for the ProductList class (cont.)

```
public Product this[int i]
{
    get
    {
        if (i < 0)
        {
          throw new
            ArgumentOutOfRangeException(i.ToString());
        }
        else if (i >= products.Count)
        {
          throw new
            ArgumentOutOfRangeException(i.ToSTring());
        }
        return products[i];
    }
    set
    {
        products[i] = value;
        Changed(this);
    }
```

The code for the ProductList class (cont.)

```
public void Save()
{
    ProductDB.SaveProducts(products);
}

public void Add(Product product)
{
    products.Add(product);
    Changed(this);
}

public void Add(string code, string description,
    decimal price)
{
    Product p = new Product(code, description, price);
    products.Add(p);
    Changed(this);
}
```

The code for the ProductList class (cont.)

```
    public void Remove(Product product)
    {
        products.Remove(product);
        Changed(this);
    }

     public static ProductList operator + (ProductList pl,
         Product p)
    {
        pl.Add(p);
        return pl;
    }

    public static ProductList operator - (ProductList pl,
        Product p)
    {
        pl.Remove(p);
        return pl;
    }

}
```

The code for the Product Maintenance form

```
private ProductList products = new ProductList();

private void frmProductMain_Load(object sender,
    System.EventArgs e)
{
    products.Changed
        += new ProductList.ChangeHandler(HandleChange);
    products.Fill();
    FillProductListBox();
}
```

Code for the Product Maintenance form (cont.)

```
private void HandleChange(ProductList products)
{
    products.Save();
    FillProductListBox();
}
```

The syntax for declaring a delegate

```
public delegate returnType DelegateName([parameterList]);
```

Code that declares a delegate in the ProductList class

```
public delegate void ChangeHandler(ProductList products);
```

A method with the same signature as the delegate

```
private static void PrintToConsole(ProductList products)
{
    Console.WriteLine("The products list has changed!");
    for (int i = 0; i < products.Count; i++)
    {
        Product p = products[i];
        Console.WriteLine(p.GetDisplayText("\t"));
    }
}
```

Code that uses a delegate

```
// create the argument that's required by the delegate
ProductList products = new ProductList();
products.Add("JSE6", "Murach's Java SE 6", 52.50m);
products.Add("CS10", "Murach's C# 2010", 54.50m);

// create the delegate and identify its method
ProductList.ChangeHandler myDelegate =
    new ProductList.ChangeHandler(PrintToConsole);

// call the delegate and pass the required argument
myDelegate(products);
```

Code that uses an anonymous method

```
ProductList.ChangeHandler myDelegate =
    delegate(ProductList products)
    {
        Console.WriteLine(
            "The products list has changed!");
        for (int i = 0; i < products.Count; i++)
        {
            Product p = products[i];
            Console.WriteLine(p.GetDisplayText("\t"));
        }
    };
myDelegate(products);
```

The syntax for declaring an event

```
public event Delegate EventName;
```

Code that declares and raises an event in the ProductList class

```
public class ProductList
{
    public delegate void ChangeHandler(
        ProductList products);
    public event ChangeHandler Changed;
                                // Declare the event

    public void Add(Product product)
    {
        products.Add(product);
        Changed(this);      // Raise the event
    }
...
}
```

Code that uses a named method to handle an event

```
ProductList products = new ProductList();

// Wire the event to the method that handles the event
products.Changed +=
    new ProductList.ChangeHandler(HandleChange);

// Handle the event
private void HandleChange(ProductList products)
{
    products.Save();
    FillProductListBox();
}
```

Code that uses an anonymous method to handle an event

```
// Wire the event to the anonymous method that handles it
products.Changed += delegate(ProductList products)
    {
        products.Save();
        FillProductListBox();
     };
```

Your Turn

- Use the Customer Maintenance solution provided
  - #11 – 14 in the exercise on 437 at the end of chapter 13.
- In Blackjack,
  - Could a deck broadcast an event when it’s empty?
  - Could a bjhand broadcast an event when it’s BUSTED?
  - Would either of those make programming the UI easier? Because you won’t have to keep checking those things in your event handlers?

Your Turn

- In Mexican Train Dominos
  - Playing a domino on a train sometimes required “flipping” of a domino.
    - What class should handle that? The train? The hand? The domino itself?
    - If the train flips a domino, how should it let the other objects (the UI) know that the domino was flipped? (And should be redisplayed?)
    - Events might have been helpful here.
- EXTRA CREDIT: If you want to try adding events and delegates I would encourage you to start with BlackJack.

What’s Next

- Next class is a “lab day”. Use the time to finish labs 1 and 2 and do a peer evaluations (and lab 1 if you haven’t done that yet).
- Reading Quiz 3
- Lab 3 – UI for BlackJack. EXTRA CREDIT (lots): UI for MTDominos.
- Quiz 1
  - Will cover inheritance, abstract classes, interfaces, delegates and events
  - Some multiple choice, a design problem and 2 “short” programming problems