Abstract: This paper describes a technique for displaying references to objects in a listbox, as opposed to presenting partial information about the objects in the listbox. The benefit of the proposed approach is that the graphical component of the application – the code in a window form class – becomes decoupled from the data associated with the objects, as well as from the operations applied to the data. The window form class is elegantly simple, because the objects stored in the listbox are constructed from a class derivation – the code that manipulates the items in the listbox is unaware of the different types of objects maintained in the container. After evaluating three design alternatives, the two-tier model is chosen to build a window form application that simulates a grocery list – the code in the presentation tier and the business tier can be extended, independent of each other. The technique uses the listbox’s DisplayMember property to display information about the objects.

Keywords: class derivation, code extensibility, event handler, inheritance diagram, listbox, object-oriented programming, reference, two-tier application, Visual C#.

Introduction

This paper describes a technique to display references to objects in a window form’s listbox, using one of the public properties of a C# class. The technique is illustrated with the Visual C# programming language; the same technique is also applicable to window forms built with the Visual Basic programming language. Both languages are supported in the Microsoft Visual Studio 2008 Integrated Development Environment.

A listbox is an object contained in a window form, and is used to display a list of values. The listbox Items collection property displays the information stored in the listbox. The items listed in the collection are of data type string.

Normally a programmer develops code that responds to the clicking of values displayed in the listbox, by implementing the SelectedIndexChanged() event handler method. Very often the data that should be stored in a listbox is much more complex than just a set of string values; in most of the cases, the listbox should store a set of objects (actually, store a set of references to objects, to be more precise). Storing objects in a listbox is advantageous because:
1. The objects are maintained in the container (the listbox itself). There is no need to access other code, artificially built into the window form application, to manufacture object data, and use it for processing.

2. The class used for creating the objects provides a public interface to manipulate the objects. The window form application only needs to know and use the public interface of the class – that is, invoke the public methods and properties of the objects to display information on the controls of the Graphical User Interface (a.k.a., GUI).
Implementation Alternatives

An example of an object that could be stored in a listbox is a food item, say a gallon of milk purchased at a grocery store. In a simple manner, this food item could be characterized as follows:

<table>
<thead>
<tr>
<th>Food Item</th>
<th>Type</th>
<th>Unit</th>
<th>Calories/Serving</th>
<th>Price/Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milk</td>
<td>Dairy</td>
<td>Gallon</td>
<td>90</td>
<td>$2.75</td>
</tr>
</tbody>
</table>

Because the listbox displays string values, programmers may use at least three approaches to present object information in a listbox:

1. Populate the listbox with the names of the objects (i.e., “Carrot”, “Milk”, “Onion”, etc.)
   a. In the window form class, create named constants to identify the attributes of each object (e.g., MILK_TYPE = “Dairy”, MILK_UNIT = “Gallon”, etc.). Similar code would have to be created for the object Carrot, Onion, and any other objects.
   b. In the listbox’s SelectedIndexChanged () event handler method, write selection statements to:
      i. First, determine the food item name (e.g., is the selected item “Milk”?).
      ii. Second, access the named constants associated with the food item, and perform any required processing – e.g., given a quantity, calculate the cost of the purchased food item.
   c. The code in the window form class needs to know the business rules to process the object data.

This design approach has serious disadvantages in terms of code maintainability. The data is tightly engraved in the window form class. The code in the event handler method becomes complex due to the multiple selection statements. It is easy to introduce logical errors when adding, updating, or deleting object data.
2. Populate the listbox with the names of the objects (e.g., “Carrot”, “Milk”, “Onion”, etc.)
   a. In the window form class, create parallel arrays (with their intrinsic fixed-size limitations), or create parallel ArrayLists, and populate each one of the containers with the attribute values of each object. For the above example, the window form class would include four additional containers; one to store the Type values, one to store the Unit values, etc.
   b. In the listbox’s SelectedIndexChanged() event handler method, use the value of the listbox’s SelectedIndex property as the index into the corresponding containers, extract the values, and perform any required processing.
   c. The code in the window form class needs to know the business rules to process the object data.

This second design approach is an enhancement over the first one; nevertheless, the partition of the object data into separate containers causes anomalies in the code:
   • First, data has to be synchronized between the listbox and its associated parallel relatives (the array(s) or the ArrayList(s)) – meaning that, addition or deletion of objects requires to maintain synchronized multiple containers.
   • Second, displaying the object’s data on the window form also forces to access multiple containers.
   • The code in the window form class is still tightly coupled with the object data.

3. Populate the listbox with references to objects. The listbox, like any C# container (array, ArrayList, ComboBox, CheckedListBox, etc.), is capable of storing references to objects. Storing references to objects implies that the design is implemented in a series of classes – one class for each type of object identified in the system. In this case, each food item is implemented in its own class (i.e., class Carrot, class Milk, class Onion, etc.). In addition, different objects that share common attributes and behaviors can be implemented in a class derivation, with a common base class at the root of the inheritance hierarchy. After storing the references to the objects in the listbox, event handler methods that need to access the objects in the container, simply obtain a reference to the base class, ignoring the actual data type (i.e., class) of the object. The code in the window form class can then treat all the objects in the listbox as if their data types were the base class, and invoke the public methods and properties of the object. For example, objects of type Carrot, Milk, or Onion can all be treated as Food, the base class of all food items. The following benefits result from this third design approach:
a. The code in the window form class becomes decoupled from the data, as well as decoupled from the operations that should be performed on the data. The food class derivation provides public properties and methods to manipulate each object. The window form class only needs to know the public interface of the food class derivation.

b. The code in the window form class is structured in a sequence (one instruction statement after another); there are no selection statements, the code is simpler.

c. All the objects are in one place, in the listbox itself – there is no need to access other constructs.

d. All the references to objects maintained in the listbox are treated in a homogenous way, regardless of the specific food item – all the objects are treated as if they were of type Food.

e. The overall design of the system is divided into two independent components or tiers:

   i. **The presentation tier** – encapsulated in the window form class. The code is responsible for serving the user requests, through the implementation of a set of event handler methods.

      • The window form class is unaware of the kind of data stored in the listbox – different kinds of foods can be added to or deleted from the listbox; the code continues to work, without any changes. The window form class simply uses the public interface of the food class derivation.

   ii. **The business tier** – implemented in the food class derivation. The classes encapsulate the data values into attributes and expose a public interface to manipulate the objects.

      • New classes (e.g., Watermelon) can be added to the system, or existing classes can be updated or deleted from the system – these changes do not affect the presentation tier, or have a minimum effect in it. Changes in the class derivation, as well as changes in the food items data (say the price changes from $2.58 to $1.49) are managed within the food class derivation.

   iii. Each tier component can be extended with new functionality, independent of each other. For example, the GUI component can be enhanced to present object data in a different manner (e.g., have a separate list displaying the grocery items currently purchased), or the data in the business tier could be obtained from a database (instead of being encoded in the classes themselves). That is, add a third database-tier. The changes derived from the proposed code extensions are managed within the tier itself.
This paper uses the **two-tier** model to demonstrate storing of references to objects in a listbox.
Methodology

When references to objects are stored in a listbox, the *Items* collection property displays the string representation of the objects, through a call to their class’ *ToString()* method. If the class does not implement its own version of method *ToString()*), the string representation of the objects is automatically obtained through a call to the inherited version of the *ToString()* method, from class *Object* – the default implementation of method *ToString()* in class *Object* returns the name of the class.

There are two alternatives to display object information in the listbox’s *Items* collection:

- Override the class’ *ToString()* method, or
- Set the listbox property *DisplayMember*\(^1\), in the properties window, during the design of the form.

This paper uses the listbox property *DisplayMember* to display object information in the *Items* collection. During design time, the listbox’s *DisplayMember* property has to be set to the name of one of the public properties of the class whose objects will be maintained in the listbox. The technique is simple and does not require adding extra code to the class (i.e., the class does not implement the *ToString()* method). For example, it is possible to set the property *DisplayMember* to a public property *FirstName*, but if such option is not satisfactory, the property *DisplayMember* can be set to a public property *LastName*, without changing the class code. The property *DisplayMember* can be set to the name of a single public property of a class. (Property *DisplayMember* cannot be set to multiple public properties of a class).

Usage of the listbox’s *DisplayMember* property is illustrated through a window form application that simulates a grocery list. The user of the form clicks one of the grocery items to view all its details. In this case, the listbox stores references to a set of objects of class *Food*, and the listbox property *DisplayMember* displays the values of *Food*’s public property *Name*.

---
\(^1\) CheckListBoxes do not support the *DisplayMember* property, but ListBoxes and ComboBoxes do support it.
The grocery list two-tier application is composed of:

1. Class *ObjectInListForm*: The window form class that displays grocery data and provides operations for: adding to, removing from, and clearing the grocery list. Class *ObjectInListForm* invokes the public interface of class *Food* to present data on the GUI.
2. The Food class derivation: *Food* is the base class for the different types of foods available at the store. The *Food* class derivation encodes the business tier of the grocery list application.

Appendix A contains:

- **The GUI design**:
  - Figure 1 identifies each one of the controls on the window form.
  - Figure 2 illustrates the listbox’s *DisplayMember* property set to the value *Name* – a public property of class *Food* – see also Figure 4.
  - Figure 3 shows the Error Provider component added to the window form. This component is used in the event handler methods associated with the clicking of the *addButton* and *deleteButton* objects, respectively.

- **The Food hierarchy derivation**:
  - The following data was used in the grocery list application:

<table>
<thead>
<tr>
<th>Food Item</th>
<th>Type</th>
<th>Unit</th>
<th>Calories/Serving</th>
<th>Price/Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>Fruit</td>
<td>Pound</td>
<td>121</td>
<td>$0.50</td>
</tr>
<tr>
<td>Carrot</td>
<td>Vegetable</td>
<td>Pound</td>
<td>27</td>
<td>$0.98</td>
</tr>
<tr>
<td>Cheese</td>
<td>Dairy</td>
<td>8-ounce</td>
<td>200</td>
<td>$4.49</td>
</tr>
<tr>
<td>Egg</td>
<td>Protein</td>
<td>Dozen</td>
<td>54</td>
<td>$1.59</td>
</tr>
<tr>
<td>Lettuce</td>
<td>Vegetable</td>
<td>Pound</td>
<td>4</td>
<td>$0.75</td>
</tr>
<tr>
<td>Milk</td>
<td>Dairy</td>
<td>Gallon</td>
<td>90</td>
<td>$2.75</td>
</tr>
<tr>
<td>New Cola</td>
<td>Beverage</td>
<td>8 fl. Oz</td>
<td>120</td>
<td>$1.25</td>
</tr>
<tr>
<td>Orange</td>
<td>Fruit</td>
<td>Pound</td>
<td>85</td>
<td>$0.80</td>
</tr>
<tr>
<td>Roast Beef</td>
<td>Protein</td>
<td>Pound</td>
<td>210</td>
<td>$10.89</td>
</tr>
<tr>
<td>Salmon</td>
<td>Fish</td>
<td>Pound</td>
<td>233</td>
<td>$15.19</td>
</tr>
<tr>
<td>Tomato</td>
<td>Fruit</td>
<td>Pound</td>
<td>27</td>
<td>$2.70</td>
</tr>
<tr>
<td>Water</td>
<td>Beverage</td>
<td>Gallon</td>
<td>0</td>
<td>$0.88</td>
</tr>
</tbody>
</table>

*Table 1*: Characterization of the food items
By inspecting the characteristics of the data, it can be seen that all of the food items have common characteristics (a.k.a., attributes); for example, Table 1 shows that all items have a name, type, unit, calories, etc. Only the values associated with the characteristics are different. It is possible then to treat all the specific food items in a general manner; that is, each item is some sort of food. Consequently, Tomato is a Food; Orange is a Food, etc. Applying class inheritance, the food item classes can all be derived from the base class Food. In the Appendix A, Figure 4 shows the class hierarchy diagram of the different types of foods. The listbox is populated with references to objects of type Food, the base class for all the classes of food items sold at Al's Friendly Market.

- In the diagram, base class Food exposes the public property Name, whose values are displayed in the listbox. Figure 2 shows the listbox's DisplayMember property being set to the value Name – the public property of class Food.

- The Implementation of the Grocery List Application. Appendix A includes the code for:
  - Class ObjectInListForm – The Presentation Tier
  - Food Class Derivation – The Business Tier
Conclusions

- This paper described a technique for displaying references to objects in a listbox. The application used the listbox’s `DisplayMember` property to display (in the `Items` collection property of the listbox) values associated with one of the object’s public properties.
  - When using the property `DisplayMember` there is no need to add extra code to the class whose objects are displayed in the listbox.
  - Overriding the default implementation of the class’ `ToString()` method is a coding alternative for controlling which object values should be displayed in the listbox.
- Out of the three design alternatives presented in this paper, the implementation of the application was based on a two-tier model. There are important code maintenance benefits associated with the model:
  - The model separates the functional responsibilities of the presentation tier (the GUI form) from the business tier (the derivation involving the food item classes).
    - The presentation tier is mainly responsible for presenting data on the GUI, in response to the user interactions with the controls displayed on the window form. The code in the window form class is disassociated from the business knowledge (i.e., the rules applicable to the processing of the data). Access to the data and operations applicable to the data is accomplished through the invocation of the public methods and properties of the objects.
    - The business tier is responsible for the processing of the data. The business knowledge is encapsulated in a set of classes – in the code example, the food class derivation encapsulates the business knowledge.
    - Changes in either one of the two tiers has no (or minimal) impact on the other tier.
  - Displaying references to objects in the listbox yields to the implementation of a truly object-oriented system. In the window form class, the code improvements become evident by inspecting some of the methods that deal with the items stored in the listbox – see the code in Appendix A. For example:
    - The event handler method `SelectedIndexChanged()` – which is responsible for displaying the details of the selected food item – simply obtains a reference to an object of type `Food` and uses its public properties to populate with values the GUI controls. The code in unaware of the actual type of object for which it displays the details – the object type could be Water, or Orange, or any other type from the food class derivation; regardless of the type, the displayed values always match those ones presented in Table 1.
The event handler method associated with the Add to Groceries List button (method `addButton_Click()`) – which is responsible for incrementing both the total cost for the purchased item and the total cost of the overall purchase – also obtains a reference to an object of type `Food`. The code is linear – devoid of selection statements – and simple; method `addButton_Click()` uses the public methods of the object to perform the calculations. The code in the window form class is disassociated from the business rules that calculate the cost of a given food item.
Appendix A – The Grocery List Application

Figure 1: ObjectInListForm window form – The GUI class

<table>
<thead>
<tr>
<th>Properties</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>groceriesListBox</td>
<td>System.Windows.Forms.ListBox</td>
</tr>
<tr>
<td>BorderStyle</td>
<td>Fixed3D</td>
</tr>
<tr>
<td>CausesValidation</td>
<td>True</td>
</tr>
<tr>
<td>ColumnWidth</td>
<td>0</td>
</tr>
<tr>
<td>ContextMenuStrip</td>
<td>(None)</td>
</tr>
<tr>
<td>Cursor</td>
<td>Default</td>
</tr>
<tr>
<td>DataSource</td>
<td>(None)</td>
</tr>
<tr>
<td>DisplayMember</td>
<td>Name</td>
</tr>
<tr>
<td>Dock</td>
<td>None</td>
</tr>
<tr>
<td>DrawMode</td>
<td>Normal</td>
</tr>
<tr>
<td>Enabled</td>
<td>True</td>
</tr>
</tbody>
</table>

Figure 2: At design time, the listbox **DisplayMember** property being set to the class **Food**'s public property **Name**.
Figure 3: The Error Provider Component added to the form.
Figure 4: Inheritance Diagram for the Food Item Classes – Food is the base class. (Due to the lack of space on the top layer, the inheritance diagram is continued on the second half of Figure 4.)
Class **ObjectInListForm** – Presentation Tier

```csharp
/*
 * Class ObjectInListForm: Code for the GUI
 */

using System;
using System.Windows.Forms;

namespace ListWithObjects
{
    public partial class ObjectInListForm : Form
    {
        double totalPurchase = 0;

        public ObjectInListForm()
        {
            InitializeComponent();
        }

        private void closeButton_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void ObjectInListForm_Load(object sender, EventArgs e)
        {
            // Populate the groceries list
            InitializeListBox();
        }

        // Display the details of the grocery-item
        private void groceriesListBox_SelectedIndexChanged(object sender, EventArgs e)
        {
            errorProvider1.Clear();
            Display((Food)groceriesListBox.Items[
                groceriesListBox.SelectedIndex]);
        }

        private void addButton_Click(object sender, EventArgs e)
        {
            errorProvider1.Clear();
            try
            {
                double quantity = Convert.ToDouble(quantityTextBox.Text);
                Food item = (Food)groceriesListBox.Items[
                    groceriesListBox.SelectedIndex];
                item.SetCost(quantity);
                costTextBox.Text = item.Cost.ToString("C");
                quantityTextBox.Clear();
                totalPurchase += item.GetPurchaseCost(quantity);
                totalTextBox.Text = totalPurchase.ToString("C");
            }
            catch (FormatException)
            {
            }
        }
    }
}
```
{  // Handle the exception associated with the  // quantity field  errorProvider1.SetError(quantityTextBox,  "Enter a number.");  quantityTextBox.Focus();  quantityTextBox.SelectAll();}
catch (ArgumentOutOfRangeException)
{
  // Handle the exception associated with the  // unselected grocery list-item  errorProvider1.SetError(groceriesListBox,  "Select a grocery item.");
}

private void deleteButton_Click(object sender, EventArgs e)
{
  errorProvider1.Clear();
  try
  {
    Food item = (Food)groceriesListBox.Items[
      groceriesListBox.SelectedIndex];
    totalPurchase -= item.Cost;
    item.ResetCost();
    costTextBox.Clear();
    quantityTextBox.Clear();
    totalTextBox.Clear();
    if (totalPurchase != 0)
      totalTextBox.Text = totalPurchase.ToString("C");
  }
  catch (ArgumentOutOfRangeException)
  {
    // Handle the exception associated with the  // unselected grocery list-item  errorProvider1.SetError(groceriesListBox,  "Select a grocery item.");
  }
}
private void clearButton_Click(object sender, EventArgs e)
{
  ClearDetails();
  InitializeListBox();
}
private void InitializeListBox()
{
  // Reset the list  groceriesListBox.Items.Clear();
  // Add object references for the fruits  groceriesListBox.Items.AddRange(new Food[
    new Banana(), new Tomato(), new Orange() ]);  // Add object references for the dairy products  groceriesListBox.Items.AddRange(new Food[
    new Milk(), new Cheese() ]);
// Add object references for the proteins
groceriesListBox.Items.AddRange(new Food[]
    { new Egg(), new RoastBeef(), new Salmon() });
// Add object references for the vegetables
groceriesListBox.Items.AddRange(new Food[]
    { new Carrot(), new Lettuce() });
// Add object references for the beverages
groceriesListBox.Items.AddRange(new Food[]
    { new NewCola(), new Water() });
// reset the total purchase variable
totalPurchase = 0;
}

// Display the object details
private void Display(Food item)
{
    caloriesTextBox.Text = item.Calories.ToString();
    costTextBox.Clear();
    typeTextBox.Text = item.Type;
    unitTextBox.Text = item.Units;
    priceTextBox.Text = item.Price.ToString("C");
    if (item.Cost != 0)
    {
        costTextBox.Text = item.Cost.ToString("C");
    }
}

private void ClearDetails()
{
    caloriesTextBox.Clear();
    costTextBox.Clear();
    priceTextBox.Clear();
    quantityTextBox.Clear();
    totalTextBox.Clear();
    typeTextBox.Clear();
    unitTextBox.Clear();
}
Food Class Derivation – Business Tier

/*
 * Class Food: Base class for the different kinds of foods.
 */

using System;

namespace ListWithObjects
{
    class Food
    {
        public Food(string name, string type, double calories, double price, string units)
        {
            Name = name;
            Type = type;
            Calories = calories;
            Price = price;
            Units = units;
        }

        public double Calories { get; private set; }
        public double Cost { get; private set; }
        public string Name { get; private set; }
        public double Price { get; private set; }
        public string Type { get; private set; }
        public string Units { get; private set; }

        public void SetCost(double quantity)
        {
            Cost += Price * quantity;
        }

        public double GetPurchaseCost(double quantity)
        {
            return quantity * Price;
        }

        public void ResetCost()
        {
            Cost = 0;
        }
    }
}
using System;
namespace ListWithObjects
{
    class Beverage : Food
    {
        private const string TYPE = "Beverage";
        public Beverage(string name, double calories, double price, string units)
        {
            base(name, TYPE, calories, price, units);
        }
    }
}

using System;
namespace ListWithObjects
{
    class Dairy : Food
    {
        private const string TYPE = "Dairy";
        public Dairy(string name, double calories, double price, string units)
        {
            base(name, TYPE, calories, price, units);
        }
    }
}

using System;
namespace ListWithObjects
{
    class Fish : Food
    {
        private const string TYPE = "Fish";
        private const string UNITS = "Pound";
        public Fish(string name, double calories, double price)
        {
            base(name, TYPE, calories, price, UNITS);
        }
    }
}
namespace ListWithObjects
{
    class Fruit : Food
    {
        private const string TYPE = "Fruit";
        private const string UNITS = "Pound";
        public Fruit(string name, double calories, double price)
            : base(name, TYPE, calories, price, UNITS)
        {
        }
    }
}

namespace ListWithObjects
{
    class Protein : Food
    {
        private const string TYPE = "Protein";
        public Protein(string name, double calories, double price, string units)
            : base(name, TYPE, calories, price, units)
        {
        }
    }
}
namespace ListWithObjects
{
    class Vegetable : Food
    {
        private const string TYPE = "Vegetable";
        public Vegetable(string name, double calories, double price,
                         string units)
            : base(name, TYPE, calories, price, units)
        {
        }
    }
}

namespace ListWithObjects
{
    class Banana : Fruit
    {
        private const string NAME = "Banana";
        private const double CALORIES = 121;
        private const double PRICE = 0.5;
        public Banana()
            : base(NAME, CALORIES, PRICE)
        {
        }
    }
}
class Carrot : Vegetable
{
    private const string NAME = "Carrot";
    private const double CALORIES = 27;
    private const double PRICE = 0.98;
    private const string UNITS = "Pound";

    public Carrot()
    : base(NAME, CALORIES, PRICE, UNITS)
    {
    }
}

class Cheese : Dairy
{
    private const string NAME = "Cheese";
    private const string UNITS = "8-Ounce";
    private const double CALORIES = 200;
    private const double PRICE = 4.49;

    public Cheese()
    : base(NAME, CALORIES, PRICE, UNITS)
    {
    }
}
/*
 * Class Egg: Represents a specific protein food. 
 *    Class Egg is derived from class Protein. 
 */

using System;

namespace ListWithObjects
{
    class Egg : Protein
    {
        private const string NAME = "Egg";
        private const string UNITS = "Dozen";
        private const double CALORIES = 54;
        private const double PRICE = 1.58;

        public Egg()
            : base(NAME, CALORIES, PRICE, UNITS)
        {
        }
    }
}

/*
 * Class Lettuce: Represents a specific vegetable food. 
 *    Class Lettuce is derived from class Vegetable. 
 */

using System;

namespace ListWithObjects
{
    class Lettuce : Vegetable
    {
        private const string NAME = "Lettuce";
        private const double CALORIES = 4;
        private const double PRICE = 0.75;
        private const string UNITS = "Pound";

        public Lettuce()
            : base(NAME, CALORIES, PRICE, UNITS)
        {
        }
    }
}
namespace ListWithObjects
{
    class Milk : Dairy
    {
        private const string NAME = "Milk";
        private const string UNITS = "Gallon";
        private const double CALORIES = 90;
        private const double PRICE = 2.75;

        public Milk()
        :
            base(NAME, CALORIES, PRICE, UNITS)
        {
        }
    }
}

namespace ListWithObjects
{
    class NewCola : Beverage
    {
        private const string NAME = "New Cola";
        private const double CALORIES = 120;
        private const double PRICE = 1.25;
        private const string UNITS = "8 fl. oz.";

        public NewCola()
        :
            base(NAME, CALORIES, PRICE, UNITS)
        {
        }
    }
}
namespace ListWithObjects
{
    class Orange : Fruit
    {
        private const string NAME = "Orange";
        private const double CALORIES = 85;
        private const double PRICE = 0.8;
        public Orange()
        :
            base(NAME, CALORIES, PRICE)
        {
        }
    }
}

namespace ListWithObjects
{
    class RoastBeef : Protein
    {
        private const string NAME = "Roast Beef";
        private const string UNITS = "Pound";
        private const double CALORIES = 210;
        private const double PRICE = 10.89;

        public RoastBeef()
        :
            base(NAME, CALORIES, PRICE, UNITS)
        {
        }
    }
}
using System;

namespace ListWithObjects
{
    class Salmon : Fish
    {
        private const string NAME = "Salmon";
        private const double CALORIES = 233;
        private const double PRICE = 15.19;
        public Salmon() : base(NAME, CALORIES, PRICE) {}
    }
}

namespace ListWithObjects
{
    class Tomato : Fruit
    {
        private const string NAME = "Tomato";
        private const double CALORIES = 27;
        private const double PRICE = 2.7;
        public Tomato() : base(NAME, CALORIES, PRICE) {}
    }
}