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. Describe the use of the Type class.

Objectives (cont.)

6. Describe the use of explicit casting when working with the objects of a base class and its derived classes.
7. Distinguish between an abstract class and a sealed class.

How inheritance works

Base class

System.Windows.Forms.Form

```
string Text
bool MinimizeBox
bool MaximizeBox

void Close()
void Show()
DialogResult ShowDialog()
```

Public properties and methods

Derived class

ProductMaintenance.frmNewProduct

```
string Text
bool MinimizeBox
bool MaximizeBox

void Close()
void Show()
DialogResult ShowDialog()
```

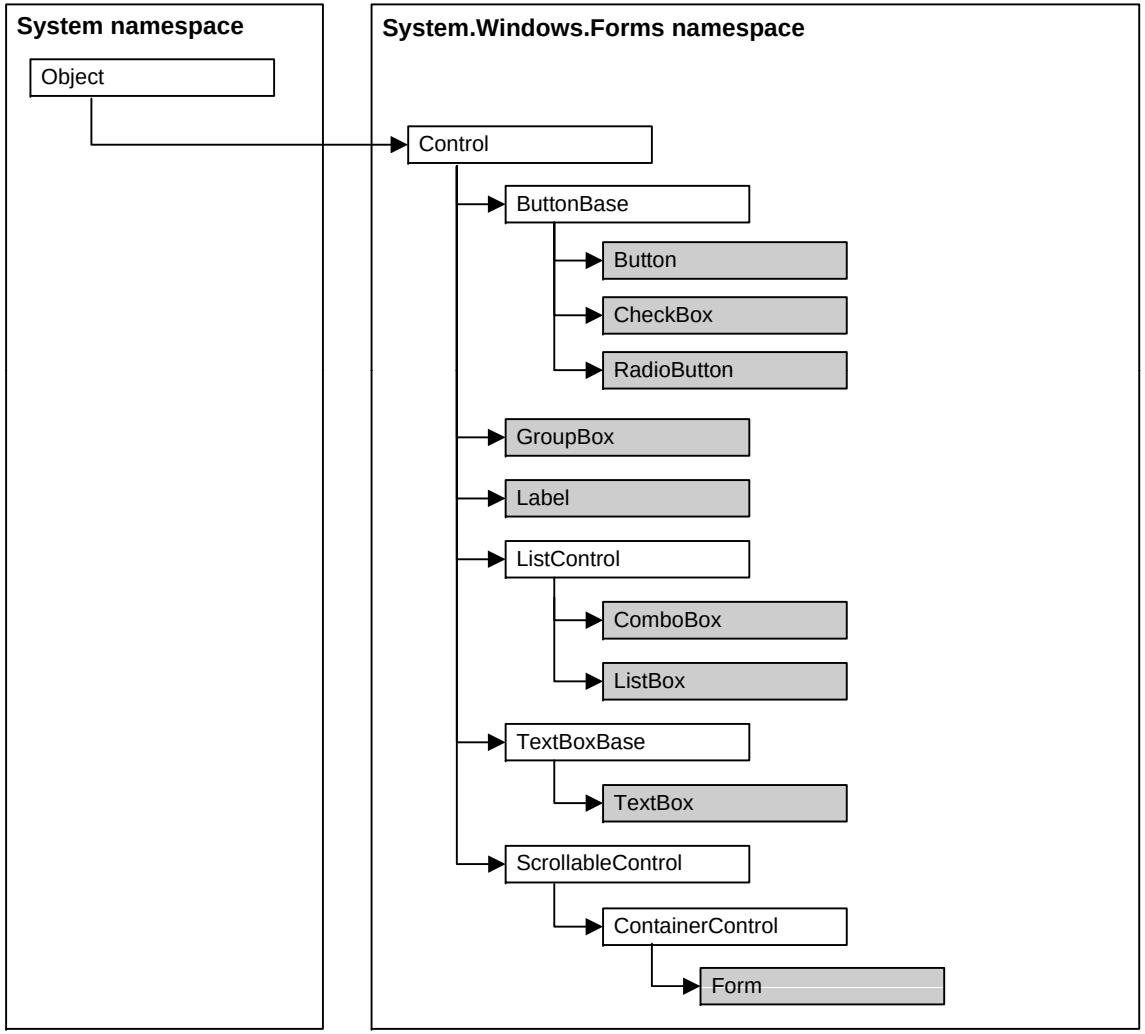
Inherited properties and methods

```
TextBox txtCode
TextBox txtDescription
TextBox txtPrice
Button btnSave
Button btnClose

Product GetNewProduct()
```

Added properties and methods

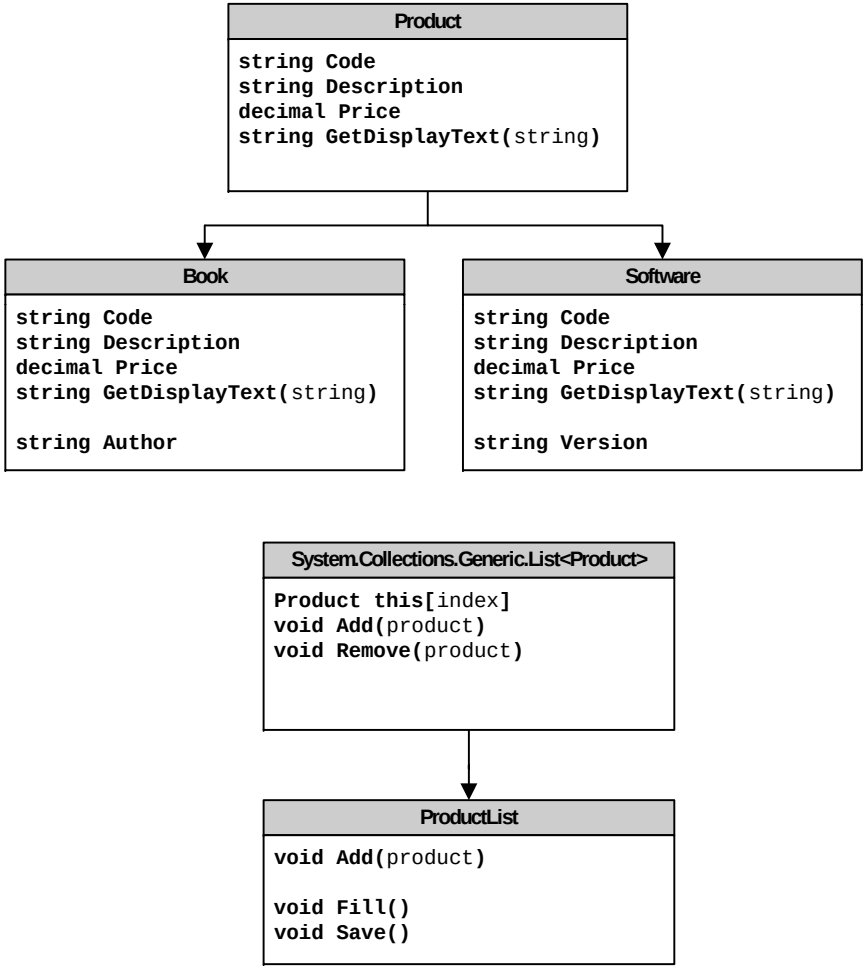
The inheritance hierarchy for form control classe



Methods of the System.Object class

```
ToString()
Equals(object)
Equals(object1, object2)
ReferenceEquals(object1, object2)
GetType()
Finalize()
MemberwiseClone()
```

Business classes for a Product Maintenance application



The code for a simplified version of the Product base class

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

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

Access modifiers

Keyword	Description
`public`	Available to all classes.
`protected`	Available only to the current class or to derived classes.
`internal`	Available only to classes in the current assembly.
`protected internal`	Available only to the current class, derived classes, or classes in the current assembly.
`private`	Available only to the containing class.

The syntax for creating subclasses

To declare a subclass

```
public class SubclassName : BaseClassName
```

To create a constructor that calls a base class constructor

```
public ClassName(parameterlist) : base(parameterlist)
```

To call a base class method or property

```
base.MethodName(parameterlist)
base.PropertyName
```

To hide a non-virtual method or property

```
public new type name
```

To override a virtual method or property

```
public override type name
```

The code for a Book class

```
public class Book : Product
{
    public string Author { get; set; }  // A new property

    public Book(string code, string description,
        string author, decimal price) : base(code,
        description, price)
    {
        this.Author = author;
               // Initializes the Author field after
    }          // the base class constructor is called.

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

Another way to override a method

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

Three versions of the GetDisplayText method

A virtual GetDisplayText method in the Product base class

```
public virtual string GetDisplayText(string sep)
{
    return code + sep + price.ToString("c") + sep
        + description;
}
```

An overridden GetDisplayText method in the Book class

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

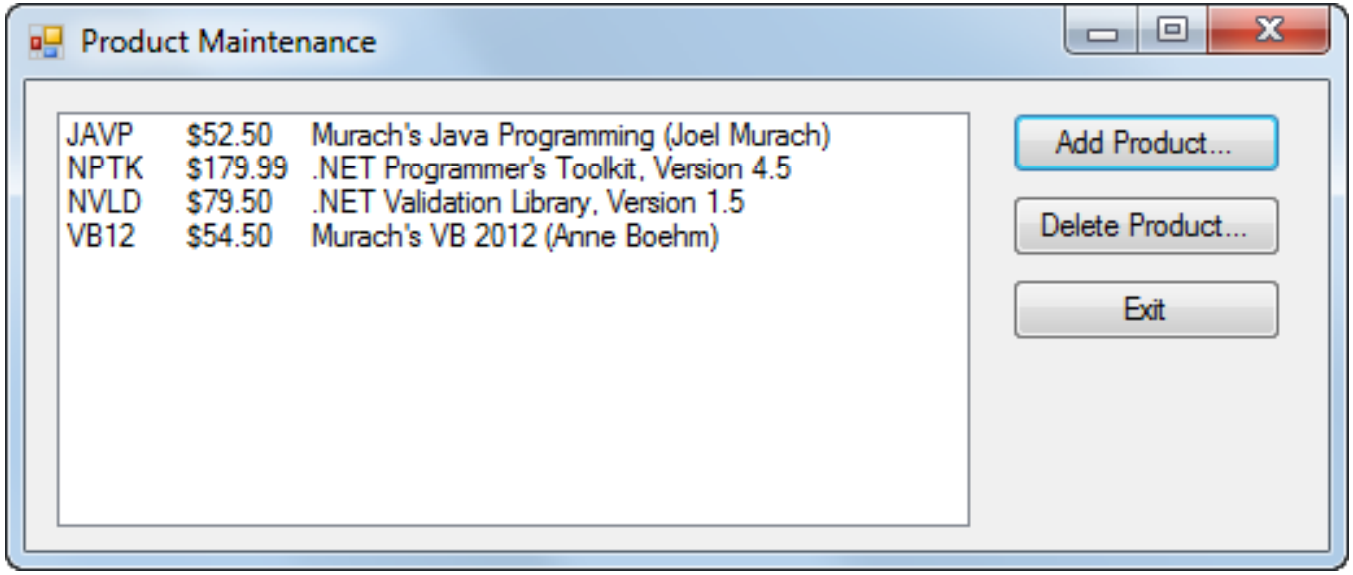
An overridden GetDisplayText method in the Software class

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

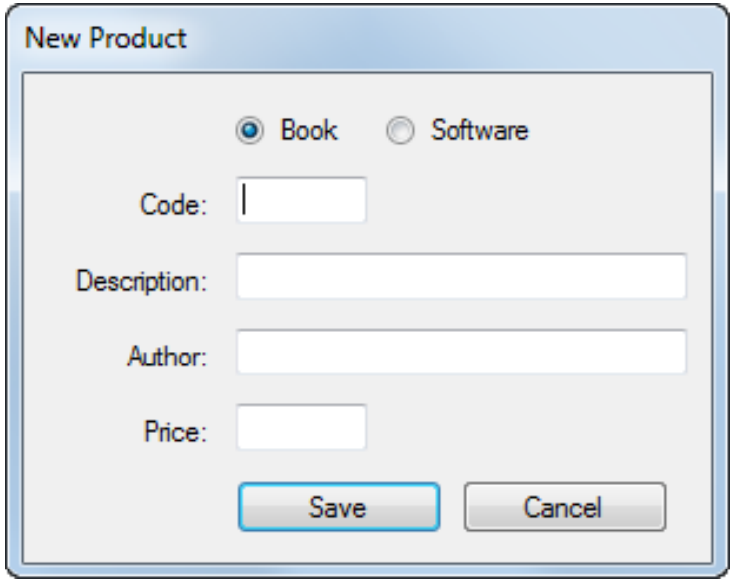
Code that uses the overridden methods

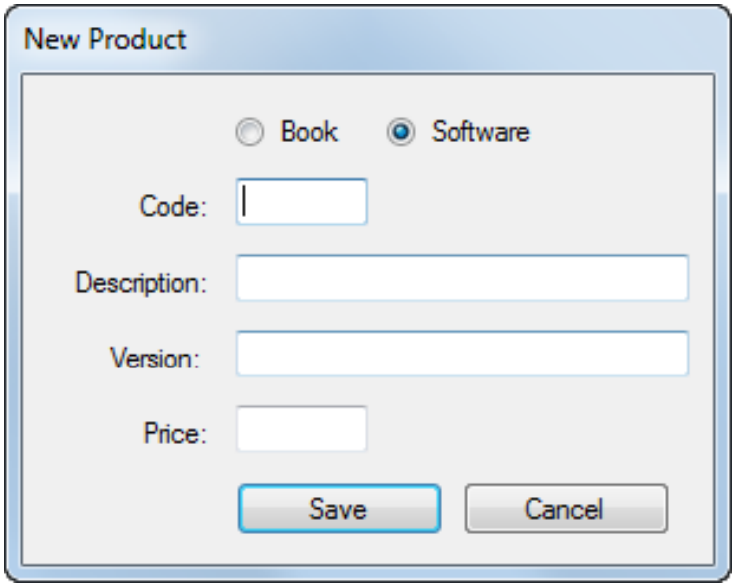
```
Book b = new Book("CS12", "Murach's C# 2012",
    "Joel Murach", 54.50m);
Software s = new Software("NPTK",
    ".NET Programmer's Toolkit", "4.5", 179.99m);
Product p;
p = b;
MessageBox.Show(p.GetDisplayText("\n"));
                        // Calls Book.GetDisplayText
p = s;
MessageBox.Show(p.GetDisplayText("\n"));
                        // Calls Software.GetDisplayText
```

The Product Maintenance form



Two versions of the New Product form





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;
    }
    set
    {
        description = value;
    }
}
```

The code for the Product class (cont.)

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

    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;
    }
```

The code for the Book class (cont.)

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

The code for the Software class (cont.)

```
    public string Version
    {
        get
        {
            return version;
        }
        set
        {
            version = value;
        }
    }

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

}
```

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);
    }
}
```

The code for the Product Maintenance form

```
public partial class frmProductMain : Form
{
    public frmProductMain()
    {
        InitializeComponent();
    }

    private ProductList products = new ProductList();

    private void frmProductMain_Load(object sender,
        System.EventArgs e)
    {
        products.Fill();
        FillProductListBox();
    }

    private void FillProductListBox()
    {
        lstProducts.Items.Clear();
        foreach (Product p in products)
            lstProducts.Items.Add(p.GetDisplayText("\t"));
    }
```

The code for the Product Maintenance form (cont.)

```
private void btnAdd_Click(object sender, System.EventArgs e)
{
    frmNewProduct newForm = new frmNewProduct();
    Product product = newForm.GetNewProduct();
    if (product != null)
    {
        products.Add(product);
        products.Save();
        FillProductListBox();
    }
}

private void btnDelete_Click(object sender, System.EventArgs e)
{
    int i = lstProducts.SelectedIndex;
    if (i != -1)
    {
        Product product = products[i];
        string message = "Are you sure you want to delete "
            + product.Description + "?";
```

The code for the Product Maintenance form (cont.)

```
            DialogResult button =
                MessageBox.Show(message, "Confirm Delete",
                MessageBoxButtons.YesNo);
            if (button == DialogResult.Yes)
            {
                products.Remove(product);
                products.Save();
                FillProductListBox();
            }
        }
    }

    private void btnExit_Click(object sender, System.EventArgs e)
    {
        this.Close();
    }
}
```

The code for the New Product form

```
public partial class frmNewProduct : Form
{
    public frmNewProduct()
    {
        InitializeComponent();
    }

    private Product product = null;

    public Product GetNewProduct()
    {
        this.ShowDialog();
        return product;
    }

    private void rbBook_CheckedChanged(object sender,
        System.EventArgs e)
    {
        if (rbBook.Checked)
        {
            lblAuthorOrVersion.Text = "Author: ";
            txtAuthorOrVersion.Tag = "Author";
        }
```

The code for the New Product form (cont.)

```
        else
        {
            lblAuthorOrVersion.Text = "Version: ";
            txtAuthorOrVersion.Tag = "Version";
        }
        txtCode.Focus();
    }
    private void btnSave_Click(object sender, System.EventArgs e)
    {
        if (IsValidData())
        {
            if (rbBook.Checked)
                product = new Book(txtCode.Text,
                    txtDescription.Text,
                    txtAuthorOrVersion.Text,
                    Convert.ToDecimal(txtPrice.Text));
            else
                product = new Software(txtCode.Text,
                    txtDescription.Text,
                    txtAuthorOrVersion.Text,
                    Convert.ToDecimal(txtPrice.Text));
            this.Close();
        }
    }
```

The code for the New Product form (cont.)

```
    private bool IsValidData()
    {
        return Validator.IsPresent(txtCode) &&
            Validator.IsPresent(txtDescription) &&
            Validator.IsPresent(txtAuthorOrVersion) &&
            Validator.IsPresent(txtPrice) &&
            Validator.IsDecimal(txtPrice);
    }

    private void btnCancel_Click(object sender, System.EventArgs e)
    {
        this.Close();
    }
}
```

The Type class

Property	Description
`Name`	Returns a string that contains the name of a type.
`FullName`	Returns a string that contains the fully qualified name of a type, which includes the namespace name and the type name.
`BaseType`	Returns a Type object that represents the class that a type inherits.
`Namespace`	Returns a string that contains the name of the namespace that contains a type.
Method	**Description**
`GetType(fullname)`	A static method that returns a Type object for the specified name.

Code that uses the Type class to get information about an object

```
Product p;
p = new Book("CS12", "Murach's C# 2012",
             "Joel Murach", 54.50m);
Type t = p.GetType();
Console.WriteLine("Name: " + t.Name);
Console.WriteLine("Namespace: " + t.Namespace);
Console.WriteLine("FullName: " + t.FullName);
Console.WriteLine("BaseType: " + t.BaseType.Name);
```

The result that's displayed on the console

```
Name: Book
Namespace: ProductMaintenance
FullName: ProductMaintenance.Book
BaseType: Product
```

How to test an object's type

```
if (p.GetType().Name == "Book")
```

Another way to test an object's type

```
if (p.GetType() ==
    Type.GetType("ProductMaintenance.Book"))
```

Two methods that display product information

```
public void DisplayProduct(Product p)
{
    MessageBox.Show(p.GetDisplayText());
}

public void DisplayBook(Book b)
{
    MessageBox.Show(b.GetDisplayText());
}
```

Code that doesn't require casting

```
Book b = new Book("CS12", "Murach's C# 2012",
                  "Joel Murach", 54.50m);
DisplayProduct(b);  // Casting is not required because Book
                    // is a subclass of Product.
```

Code that requires casting

```
Product p = new Book("CS12", "Murach's C# 2012",
                     "Joel Murach", 54.50m);
DisplayBook((Book)p);  // Casting is required because
                       // DisplayBook accepts a Book object.
```

Code that throws a casting exception

```
Product p = new Software("NPTK",
    ".NET Programmer's Toolkit", "4.5", 179.99m);
DisplayBook((Book)p);    // Will throw a casting exception
                         // because p is a Software object,
                         // not a Book object.
```

Code that uses the as operator

```
Product p = new Book("CS12", "Murach's C# 2012",
                     "Joel Murach", 54.50m);
DisplaySoftware(p as Software);  // Passes null because p
                                 // isn't a Software object
```

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");
    }
}
```

The class declaration for a sealed Book class

```
public sealed class Book : Product
```

How sealed methods work

A base class named A that declares a virtual method

```
public class A
{
    public virtual void ShowMessage()
    {
        MessageBox.Show("Hello from class A");
    }
}
```

A class named B that inherits class A and overrides and seals its method

```
public class B : A
{
    public sealed override void ShowMessage()
    {
        MessageBox.Show("Hello from class B");
    }
}
```

How sealed methods work (cont.)

A class named C that inherits class B and tries to override its sealed method

```
public class C : B
{
    public override void ShowMessage()   // Not allowed
    {
        MessageBox.Show("Hello from class C");
    }
}
```