How to work with indexers, delegates, events, and operators

The code for a simple ProductList class

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

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

    public int Count => products.Count;

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

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

The code for a simple ProductList class (cont.)

```
    public Product GetProductByIndex(int i) =>  products[i];

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

    public void Fill() => products = ProductDB.GetProducts();

    public void Save() => ProductDB.SaveProducts(products);

}
```

The ProductList class

Constructor	Description
()	Creates a new product list.
Indexer	**Description**
[index**]**	Provides access to the product at the specified position.
[code**]**	Provides access to the product with the specified code.
Property	**Description**
Count	An integer that indicates how many Product objects are in the list.
Method	**Description**
Add(product**)**	Adds the specified Product object to the list.

The ProductList class (cont.)

Method	Description
`Add(code, description, price)`	Creates a Product object with the specified code, description, and price values, and then adds the Product object to the list.
`Remove(product)`	Removes the specified Product object from the list.
`Fill()`	Fills the list with product data from a file.
`Save()`	Saves the products to a file.

The ProductList class (cont.)

Operator	Description
`+`	Adds a Product object to the list.
`-`	Removes a Product object from the list.
Delegate	**Description**
`ChangeHandler`	Can be used to register the method that's used to handle the Changed event.
Event	**Description**
`Changed`	Raised whenever a Product object is added to, updated in, or removed from the list.

An indexer that uses an integer as an index

```
private List<Product> products;

public Product this[int i]
{
    get
    {
        return products[i];
    }
    set
    {
        products[i] = value;
    }
}
```

A read-only indexer that uses a string as an index

```
public Product this[string code]
{
    get
    {
        foreach (Product p in products)
        {
            if (p.Code == code)
                return p;
        }
        return null;
    }
}
```

A read-only indexer with an expression body

```
public Product this[int i] => products[i];
```

Code that uses these indexers

```
ProductList products = new ProductList();
products.Add("CS15", "Murach's C# 2015", 56.50m);
Product p1 = products[0];
Product p2 = products["CS15"];
products[i] = new Product(code, description, price);
```

An indexer that checks the range and throws an argument exception

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

An indexer that validates data and throws an argument exception

```
public Product this[string code]
{
    get
    {
        if (code.Length > 4)
        {
            throw new ArgumentException(
                "Maximum length of Code is 4 characters.");
        }
        foreach (Product p in products)
        {
            if (p.Code == code)
                return p;
        }
        return null;
    }
}
```

Three argument exceptions

Exception	Description
`ArgumentOutOfRangeException(message)`	Use when the value is outside the acceptable range of values.
`ArgumentNullException(message)`	Use when the value is null and a null value is not allowed.
`ArgumentException(message)`	Use when the value is invalid for any other reason.

An if statement that validates data before setting a property value

```
Product p = null;
if (txtCode.Text.Length <= 4)
     p = products[txtCode.Text];
```

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);
```

Code in a form that uses the delegate

```
public partial class frmProducts : Form
{
    // create the delegate and identify the method it uses
    ProductList.ChangeHandler myDelegate =
        new ProductList.ChangeHandler(PrintToConsole);

    // 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 in a form that uses the delegate (cont.)

```
    private void frmProducts_Load(object sender, EventArgs e)
    {
        // create the argument that's required by the delegate
        ProductList products = new ProductList();

        // add products to the product list
        products.Add("BJWN",
            "Murach's Beginning Java with NetBeans", 57.50m);
        products.Add("CS15", "Murach's C# 2015", 56.50m);

        // call the delegate and pass the required argument
        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 in a form that wires the event handler and handles the event

```
ProductList products = new ProductList();

private void frmProducts_Load(object sender, System.EventArgs e)
{
    // wire the event to the method that handles the event
    products.Changed +=
        new ProductList.ChangeHandler(PrintToConsole);
    ...
}

// the method that handles the event
private 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"));
    }
}
```

How to create a delegate using an anonymous method

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

How to create a delegate using a lambda expression

```
ProductList.ChangeHandler myDelegate = products =>
{
    Console.WriteLine("The products list has changed!");
    for (int i = 0; i < products.Count; i++) { ... }
};
myDelegate(products);
```

How wire an event using an anonymous method

```
products.Changed += delegate (ProductList products)
{
    Console.WriteLine("The products list has changed!");
    for (int i = 0; i < products.Count; i++) { ... }
};
```

How wire an event using a lambda expression

```
products.Changed += products =>
{
    Console.WriteLine("The products list has changed!");
    for (int i = 0; i < products.Count; i++) { ... }
};
```

The syntax for overloading unary operators

```
public static returnType operator
    unary-operator(type operand)
```

The syntax for overloading binary operators

```
public static returnType operator
    binary-operator(type-1 operand-1, type-2 operand-2)
```

Common operators you can overload

Unary operators

```
+ - ! ++ -- true false
```

Binary operators

```
+ - * / % & | == != > <  >= <=
```

The Equals method of the Object class

Method	Description
`Equals(object)`	Returns a Boolean value that indicates whether the current object refers to the same instance as the specified object.
`Equals(object1, object2)`	A static version of the Equals method that compares two objects to determine if they refer to the same instance.

The GetHashCode method of the Object class

Method	Description
`GetHashCode()`	Returns an integer value that’s used to identify objects in a hash table. If you override the Equals method, you must also override the GetHashCode method.

Part of a ProductList class that overloads the + operator

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

    public void Add(Product p)
    {
        products.Add(p);
    }

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

Code that uses the + operator of the ProductList class

```
ProductList products = new ProductList();
Product p = new Product("CS15", "Murach's C# 2015",
                        56.50m);
products = products + p;
```

Another way to use the + operator

```
products += p;
```

Code that uses an expression-bodied operator

```
public ProductList Add(Product p)
{
    products.Add(p);
    return this;
}

public static ProductList operator + (
    ProductList pl, Product p) => pl.Add(p);
```

Code that overloads the == operator for a Product class

```
public static bool operator == (Product p1, Product p2)
{
    if (Object.Equals(p1, null))
        if (Object.Equals(p2, null))
            return true;
        else
            return false;
    else
        return p1.Equals(p2);
}

public static bool operator != (Product p1, Product p2)
{
    return !(p1 == p2);
}
```

Code that overloads the == operator for a Product class (cont.)

```
public override bool Equals(Object obj)
{
    if (obj == null)
        return false;
    Product p = (Product)obj;
    if (this.Code == p.Code &&
        this.Description == p.Description &&
        this.Price == p.Price)
        return true;
    else
        return false;
}

public override int GetHashCode()
{
    string hashString = this.Code + this.Description
                              + this.Price;
    return hashString.GetHashCode();
}
```

Code that uses the == operator of the Product class

```
Product p1 = new Product("CS15", "Murach's C# 2015", 56.50m);
Product p2 = new Product("CS15", "Murach's C# 2015", 56.50m);
if (p1 == p2)        // This evaluates to true. Without the
                     // overloaded == operator, it would
                     // evaluate to false.
```

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 => 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 Product this[string code]
{
    get
    {
        foreach (Product p in products)
        {
            if (p.Code == code)
                return p;
        }
        return null;
    }
}
public void Fill() => products = ProductDB.GetProducts();

public void Save() => ProductDB.SaveProducts(products);
```

The code for the ProductList class (cont.)

```
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);
}

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

The code for the ProductList class (cont.)

```
    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, EventArgs e)
{
    products.Changed += new ProductList.ChangeHandler(HandleChange);
    products.Fill();
    FillProductListBox();
}

private void FillProductListBox()
{
    Product p;
    lstProducts.Items.Clear();
    for (int i = 0; i < products.Count; i++)
    {
        p = products[i];
        lstProducts.Items.Add(p.GetDisplayText("\t"));
    }
}
```

The code for the Product Maintenance form (cont.)

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

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

The code for the Product Maintenance form (cont.)

```
            if (button == DialogResult.Yes)
            {
                products -= product;
            }
        }
    }

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

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