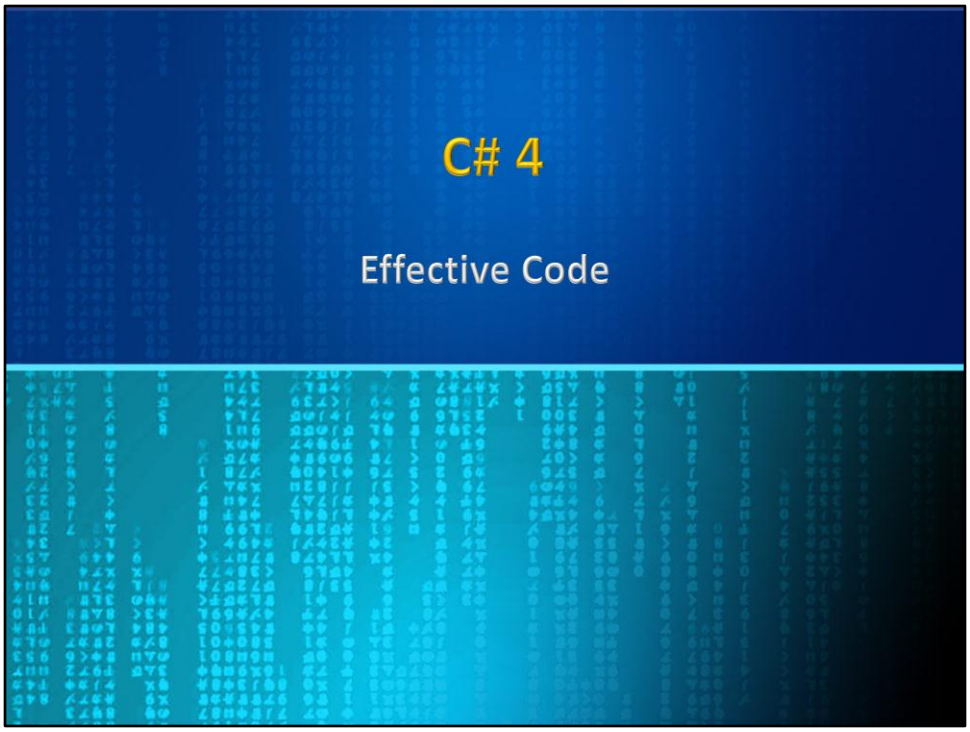
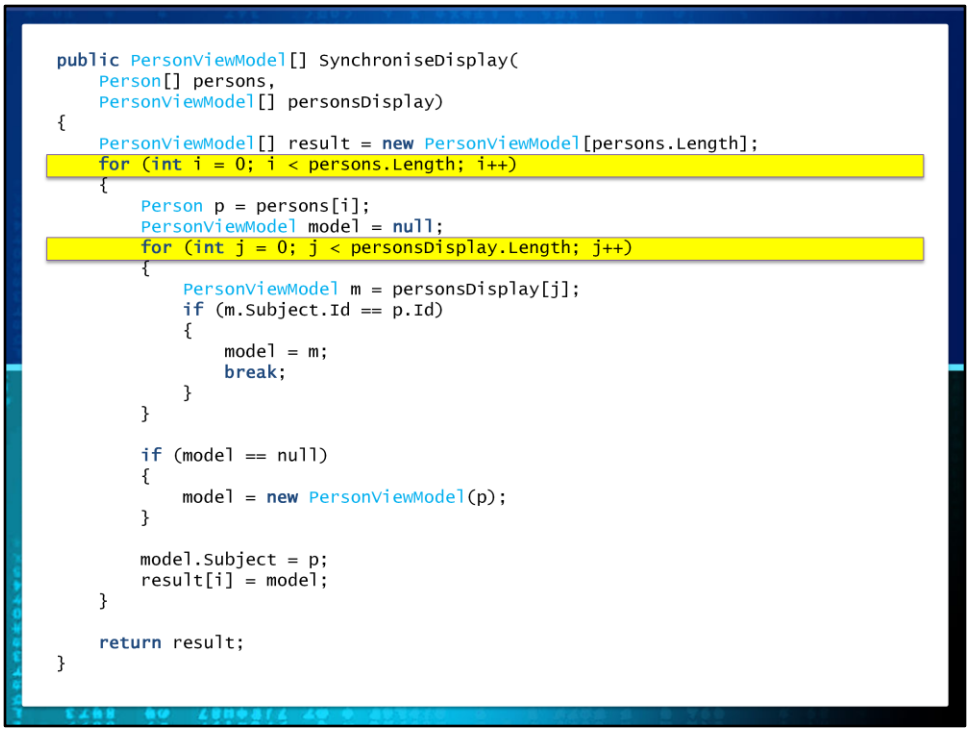

C# 4
Effective Code

How did we get here?

THE PATH TO C# 4

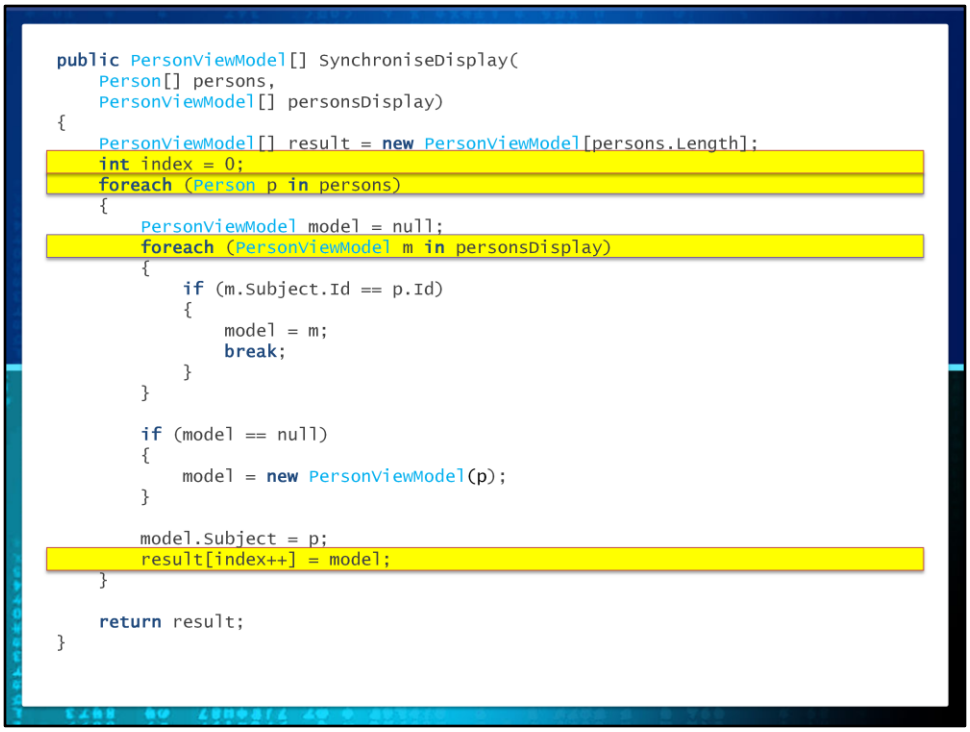


```
public PersonViewModel[] SynchroniseDisplay(
    Person[] persons,
    PersonViewModel[] personsDisplay)
{
    PersonViewModel[] result = new PersonViewModel[persons.Length];
    for (int i = 0; i < persons.Length; i++)
    {
        Person p = persons[i];
        PersonViewModel model = null;
        for (int j = 0; j < personsDisplay.Length; j++)
        {
            PersonViewModel m = personsDisplay[j];
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        if (model == null)
        {
            model = new PersonViewModel(p);
        }

        model.Subject = p;
        result[i] = model;
    }

    return result;
}
```

Here's some code pulled out of a project and rewritten in a C/C++ style – as if someone writing C would do a MVVW application …

Don't worry too much about what the code does – I'll highlight the important bits as we go.

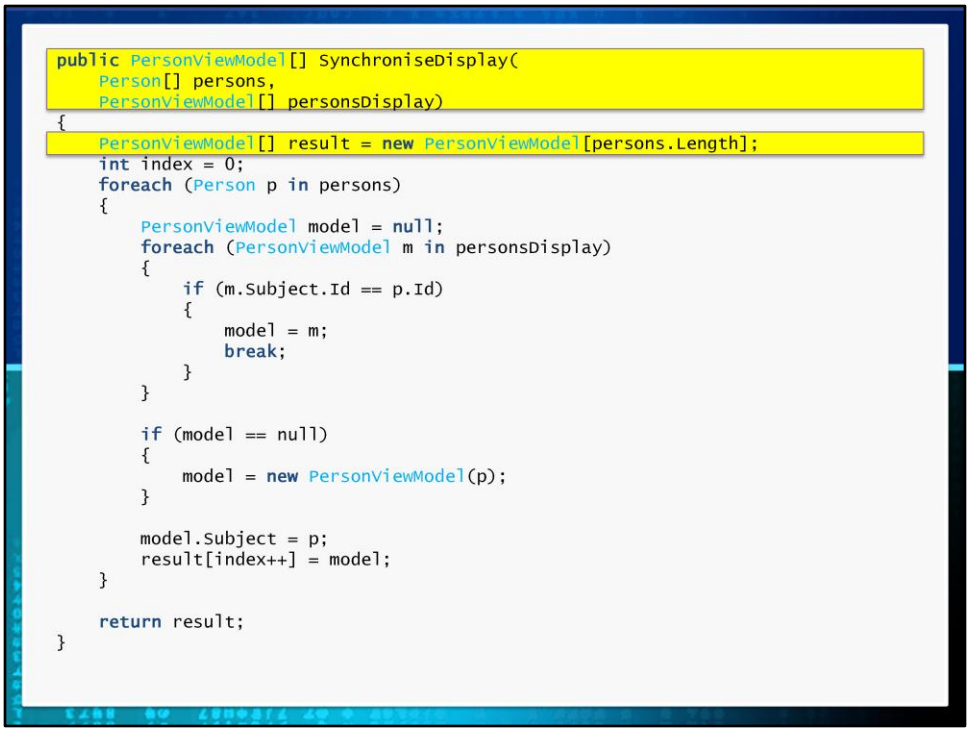
Given a list of Person instances and a list of existing PersonViewModel instances, return a list of ViewModels that matches the list of Person. Reuse any existing ViewModels that match existing Persons, making new ViewModels as necessary.



```
public PersonViewModel[] SynchroniseDisplay(
    Person[] persons,
    PersonViewModel[] personsDisplay)
{
    PersonViewModel[] result = new PersonViewModel[persons.Length];
    int index = 0;
    foreach (Person p in persons)
    {
        PersonViewModel model = null;
        foreach (PersonViewModel m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        if (model == null)
        {
            model = new PersonViewModel(p);
        }

        model.Subject = p;
        result[index++] = model;
    }

    return result;
}
```

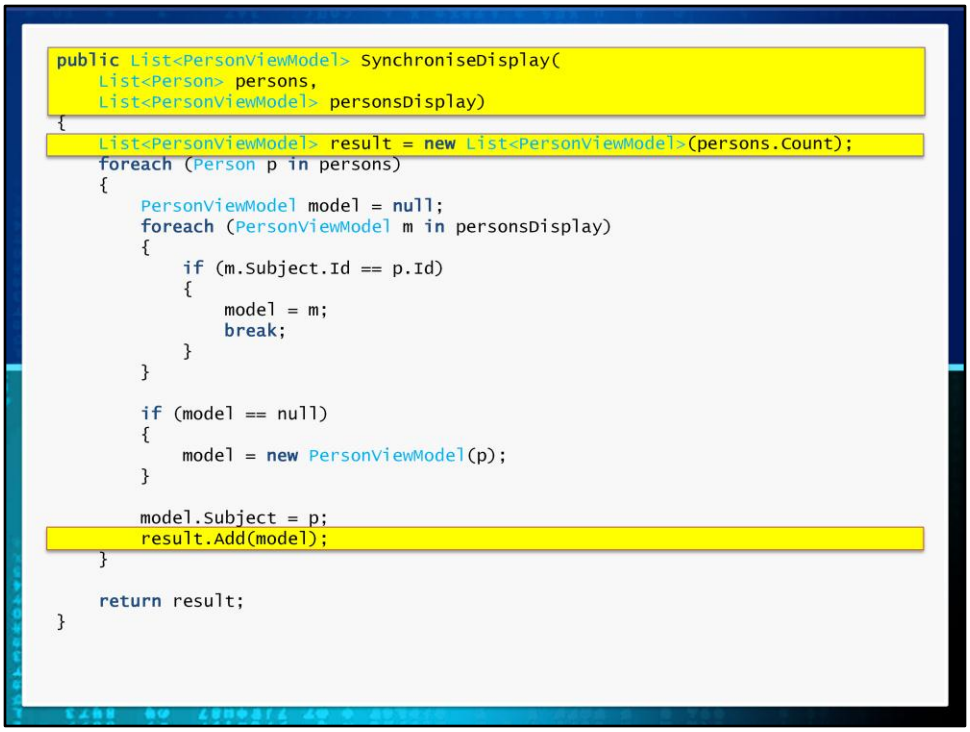
Here's some code pulled out of a project and rewritten in a C/C++ style – as if someone writing C would do a MVVW application …

Given a list of Person instances and a list of existing PersonViewModel instances, return a list of ViewModels that matches the list of Person. Reuse any existing ViewModels that match existing Persons, making new ViewModels as necessary.



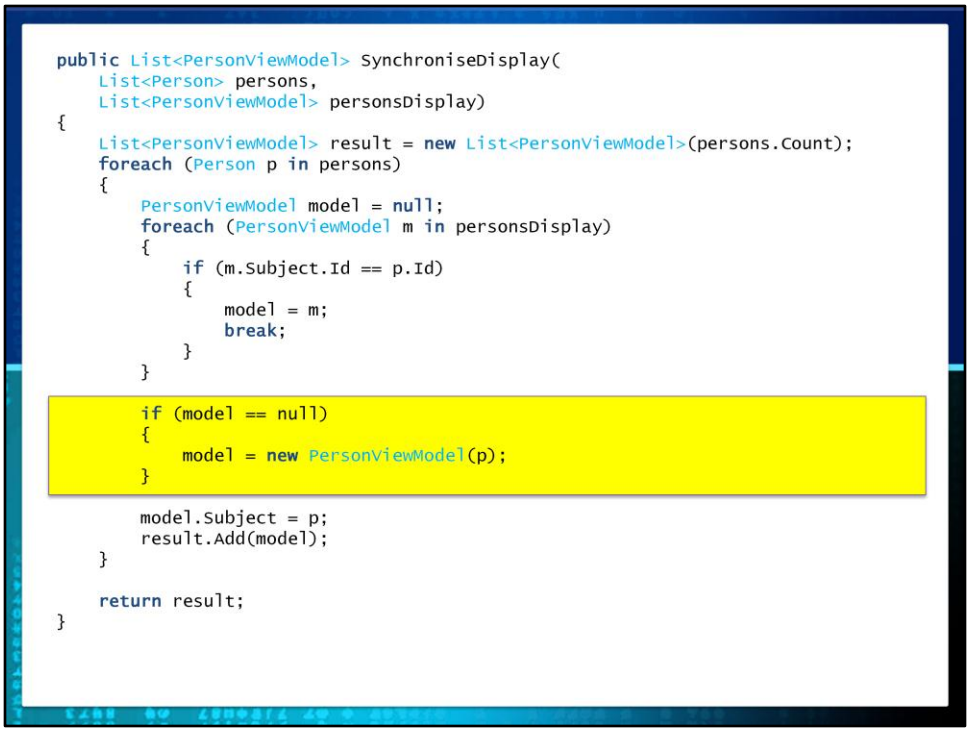
```
public PersonViewModel[] SynchroniseDisplay(
    Person[] persons,
    PersonViewModel[] personsDisplay)
{
    PersonViewModel[] result = new PersonViewModel[persons.Length];
    int index = 0;
    foreach (Person p in persons)
    {
        PersonViewModel model = null;
        foreach (PersonViewModel m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        if (model == null)
        {
            model = new PersonViewModel(p);
        }

        model.Subject = p;
        result[index++] = model;
    }

    return result;
}
```

In C# 2.0, we gained generic lists – so we could move away from the awkwardness of arrays



```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    List<PersonViewModel> result = new List<PersonViewModel>(persons.Count);
    foreach (Person p in persons)
    {
        PersonViewModel model = null;
        foreach (PersonViewModel m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        if (model == null)
        {
            model = new PersonViewModel(p);
        }

        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don’t need to manage a separate index any more.

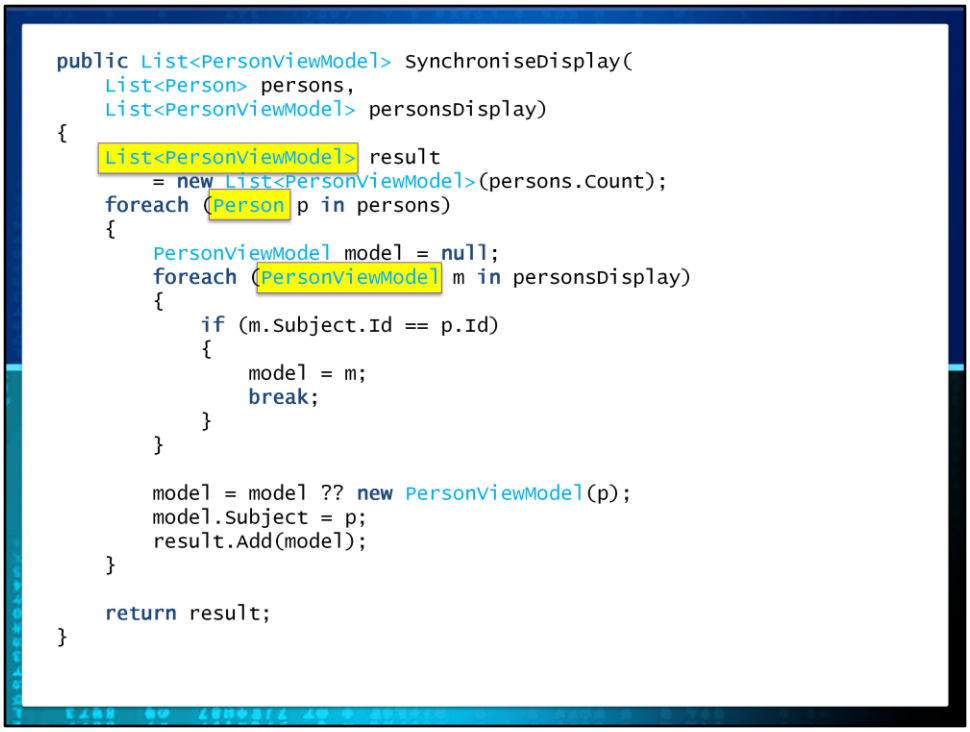


```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    List<PersonViewModel> result = new List<PersonViewModel>(persons.Count);
    foreach (Person p in persons)
    {
        PersonViewModel model = null;
        foreach (PersonViewModel m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        if (model == null)
        {
            model = new PersonViewModel(p);
        }

        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don’t need to manage a separate index any more.

```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    List<PersonViewModel> result
        = new List<PersonViewModel>(persons.Count);
    foreach (Person p in persons)
    {
        PersonViewModel model = null;
        foreach (PersonViewModel m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.

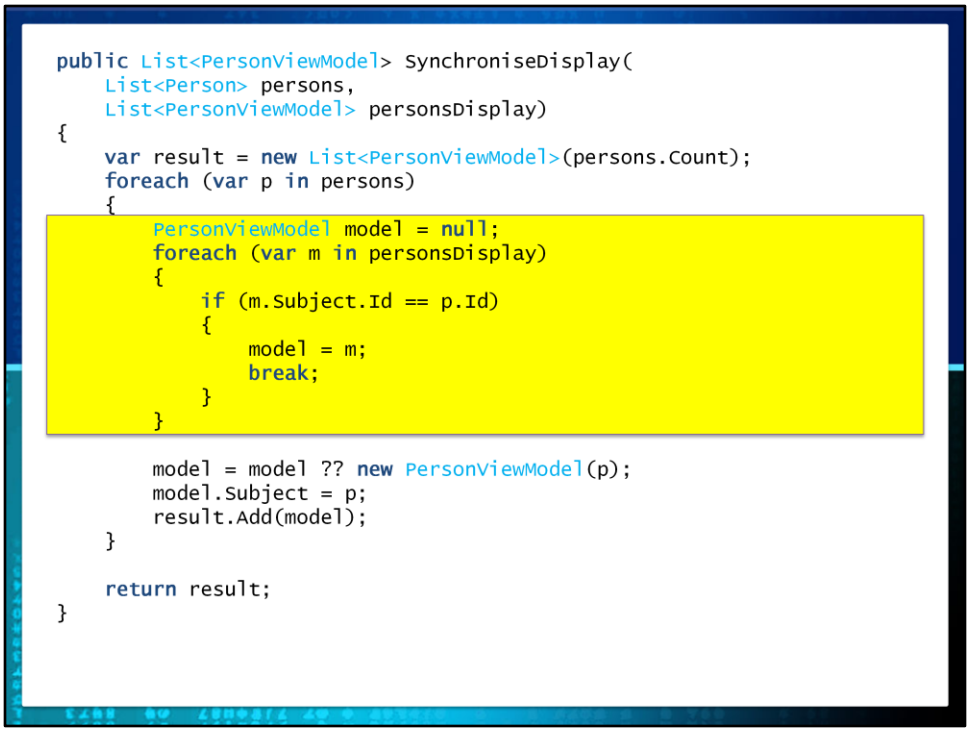


```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    List<PersonViewModel> result
        = new List<PersonViewModel>(persons.Count);
    foreach (Person p in persons)
    {
        PersonViewModel model = null;
        foreach (PersonViewModel m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.

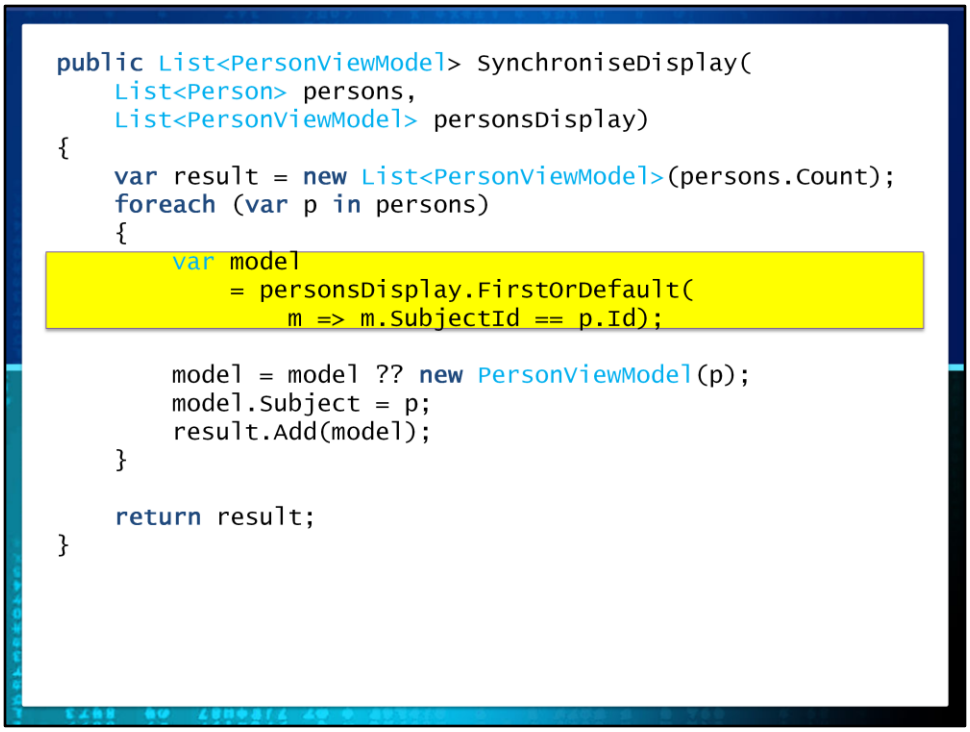
```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    var result = new List<PersonViewModel>(persons.Count);
    foreach (var p in persons)
    {
        PersonViewModel model = null;
        foreach (var m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.



```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    var result = new List<PersonViewModel>(persons.Count);
    foreach (var p in persons)
    {
        PersonViewModel model = null;
        foreach (var m in personsDisplay)
        {
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.



```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    var result = new List<PersonViewModel>(persons.Count);
    foreach (var p in persons)
    {
        var model
            = personsDisplay.FirstOrDefault(
                m => m.SubjectId == p.Id);

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.

```
public List<PersonViewModel> SynchroniseDisplay(
    List<Person> persons,
    List<PersonViewModel> personsDisplay)
{
    var result = new List<PersonViewModel>(persons.Count);
    foreach (var p in persons)
    {
        var model
            = personsDisplay.FirstOrDefault(
                m => m.Subject.Id == p.Id);

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.

```
public IEnumerable<PersonViewModel> SynchroniseDisplay(
    IEnumerable<Person> persons,
    IEnumerable<PersonViewModel> personsDisplay)
{
    var result
        = new List<PersonViewModel>(persons.Count());

    foreach (var p in persons)
    {
        var model
            = personsDisplay.FirstOrDefault(
                m => m.Subject.Id == p.Id);

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.

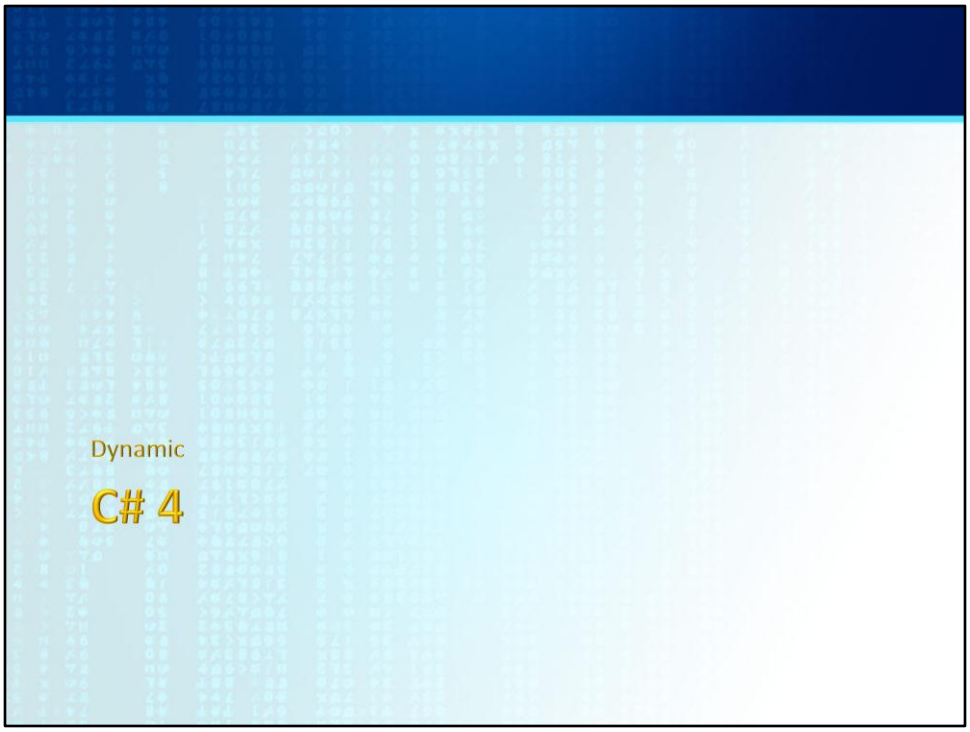
```
public IEnumerable<PersonViewModel> SynchroniseDisplay(
    IEnumerable<Person> persons,
    IEnumerable<PersonViewModel> personsDisplay)
{
    var result
        = new List<PersonViewModel>(persons.Count());

    foreach (var p in persons)
    {
        var model
            = personsDisplay.FirstOrDefault(
                m => m.Subject.Id == p.Id);

        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```

Changed the arrays to generic lists. Note that we don't need to manage a separate index any more.

Then and Now …

```
public PersonViewModel[] SynchroniseDisplay(
    Person[] persons,
    PersonViewModel[] personsDisplay)
{
    PersonViewModel[] result = new PersonViewModel[persons.Length];
    for (int i = 0; i < persons.Length; i++)
    {
        Person p = persons[i];
        PersonViewModel model = null;
        for (int j = 0; j < personsDisplay.Length; j++)
        {
            PersonViewModel m = personsDisplay[j];
            if (m.Subject.Id == p.Id)
            {
                model = m;
                break;
            }
        }

        if (model == null)
        {
            model = new PersonViewModel(p);
        }

        model.Subject = p;
        result[i] = model;
    }

    return result;
}
```

```
public IEnumerable<PersonViewModel> SynchroniseDisplay(
    IEnumerable<Person> persons,
    IEnumerable<PersonViewModel> personsDisplay)
{
    var result
        = new List<PersonViewModel>(persons.Count());

    foreach (var p in persons)
    {
        var model
            = personsDisplay.FirstOrDefault(
                m => m.Subject.Id == p.Id);
        model = model ?? new PersonViewModel(p);
        model.Subject = p;
        result.Add(model);
    }

    return result;
}
```


Dynamic
C# 4

dynamic

some
A way to opt-out of compile time checks

"Trust me,
I know what I'm doing"

Calling a Method

```
object calc = GetCalculator();
Type calcType = calc.GetType();
object res
    = calcType.InvokeMember(
        "Add",
        BindingFlags.InvokeMethod,
        null,
        new object[] { 10, 20 });
int sum = Convert.ToInt32(res);
```

```
dynamic calc = GetCalculator();
int sum = calc.Add(10, 20);
```

Reflection vs Dynamic

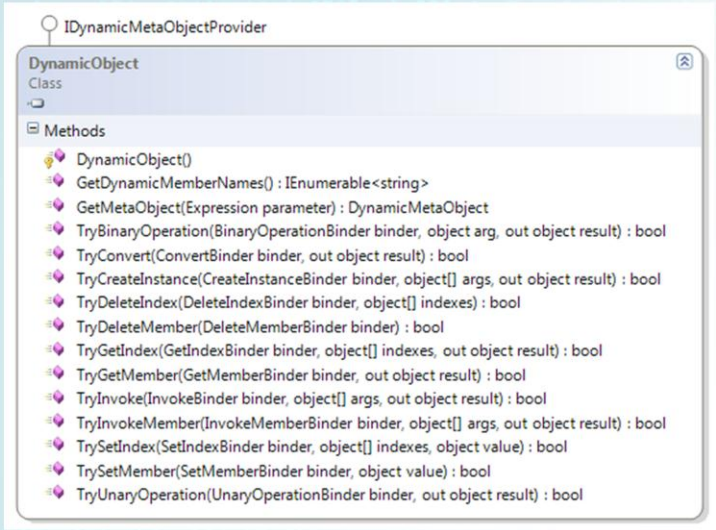
Dynamic

```
public void Speak()
{
    object critter = FindCritter();
    critter.Speak();
}
```

```
public void Speak()
{
    IAnimal critter = FindCritter();
    critter.Speak();
}
```

```
public void Speak()
{
    dynamic critter = FindCritter();
    critter.Speak();
}
```

Getting involved



Dynamic

```
// First open a connection to our database
dynamic db = OpenDatabaseConnection("sample");

// Query for Smiths
var persons = db.FindByFamilyName<Person>("Smith");
foreach (var person in persons)
{
    // Process a person
}


            db.FindByKnownAs<Person>("John");
            db.FindBySuburb<Address>("Kelson");
            db.FindByCode<Organisation>("BNZ");
```

A hypothetical example …
The db object would descend from DynamicObject and can choose how to handle the method we call, what ever it might be.

For a real world example, check out Simple.Data online.

Optional & Named Parameters

C# 4

Excel ChartWizard

```
xlChart.ChartWizard(
    cellRange.CurrentRegion,
    Constants.xl3DBar,
    Type.Missing,
    Excel.XlRowCol.xlColumns,
    1,
    2,
    false,
    xlSheet.Name,
    Type.Missing,
    Type.Missing,
    Type.Missing);
```

```
xlChart.ChartWizard(
    cellRange.CurrentRegion,
    Constants.xl3DBar,
    PlotBy: Excel.XlRowCol.xlColumns,
    SeriesLabels: 2,
    CategoryLabels: 1,
    HasLegend: false,
    Title: xlSheet.Name);
```

Code Clarity

```
dialog.Display(model, true, false, true, false);
```

```
public void Display(
    ViewModel model,
    bool modal,
    bool resizable,
    bool showHeader,
    bool allowApply)
{
    // ...
}
```

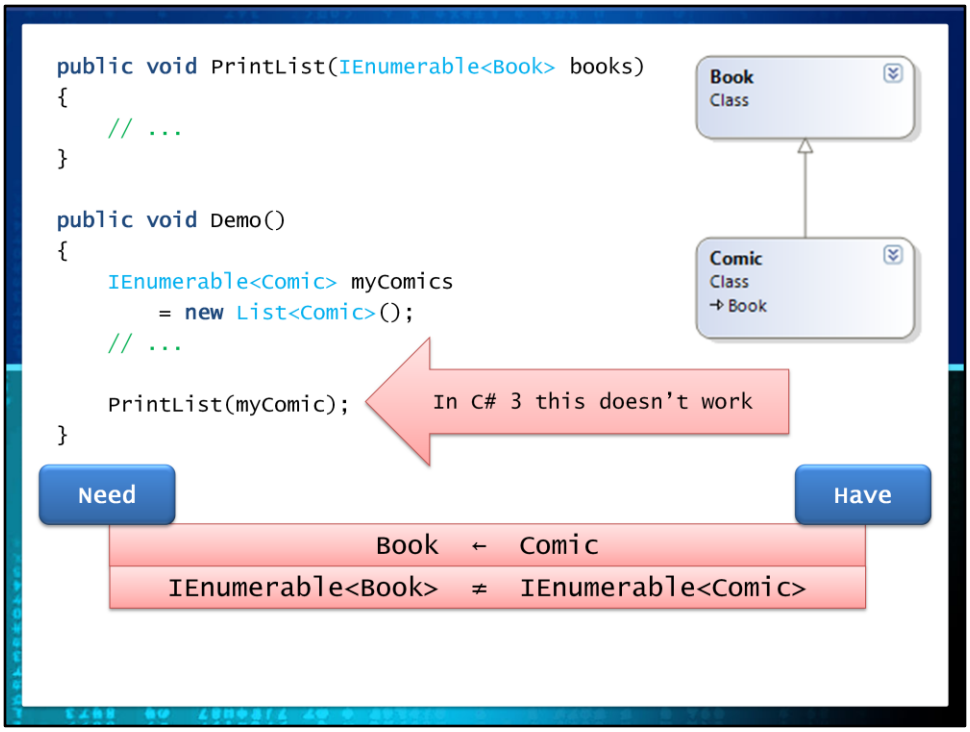
```
dialog.Display(
    model,
    modal: true,
    resizable: false,
    showHeader: true,
    allowApply: false);
```

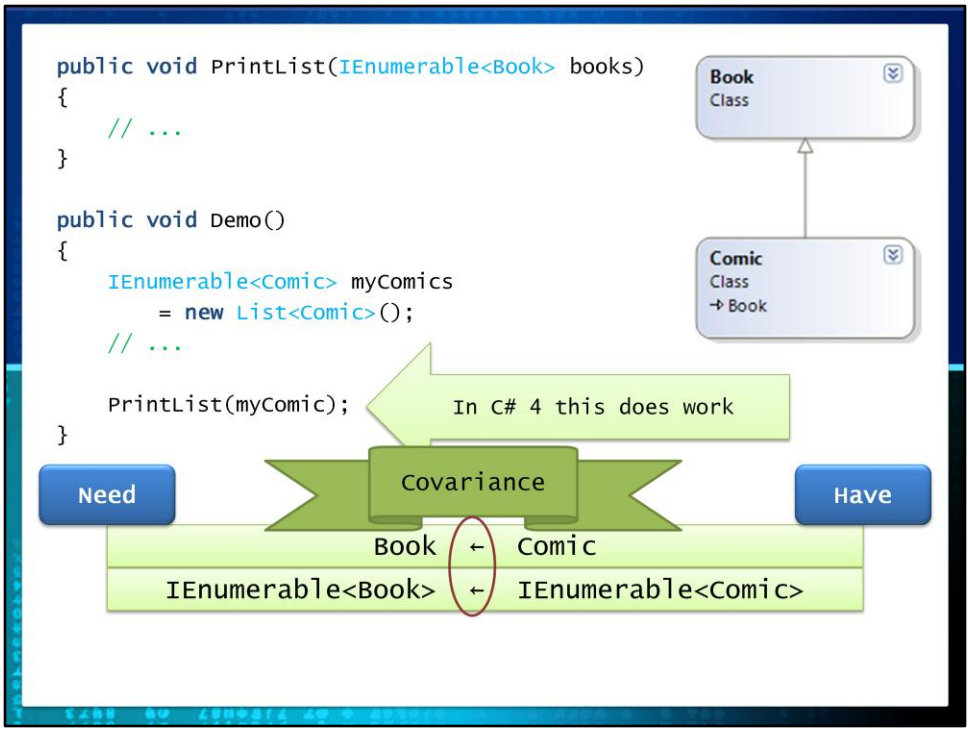
```
public void Display(
    ViewModel model,
    bool modal = true,
    bool resizable = true,
    bool showHeader = true,
    bool allowApply = true)
{
    // ...
}
```

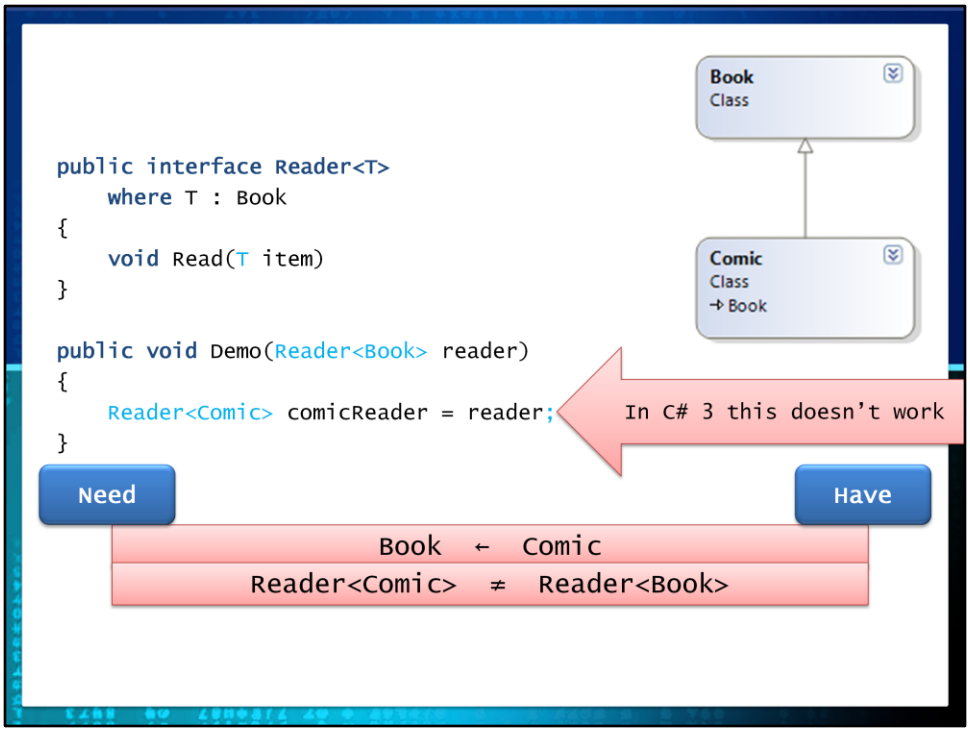
```
dialog.Display(
    model,
    resizable: false,
    allowApply: false);
```

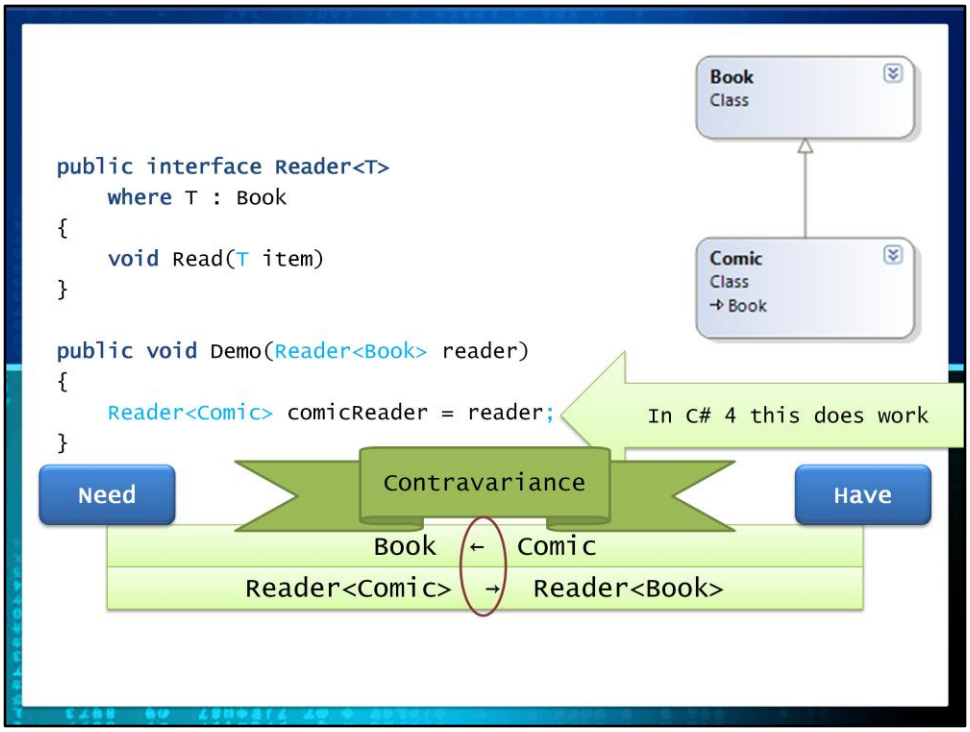
Covariance and Contravariance

C# 4


public void PrintList(IEnumerable<Book> books)
{
// ...
}
public void Demo()
{
IEnumerable<Comic> myComics
= new List<Comic>();
// ...
PrintList(myComic);
}
Book
Class
Comic
Class
→ Book
In C# 3 this doesn't work
Need
Have
Book ← Comic
IEnumerable<Book> ≠ IEnumerable<Comic>


public void PrintList(IEnumerable<Book> books)
{
// ...
}
public void Demo()
{
IEnumerable<Comic> myComics
= new List<Comic>();
// ...
PrintList(myComic);
}
Book
Class
Comic
Class
→ Book
In C# 4 this does work
Need
Covariance
Have
Book ← Comic
IEnumerable<Book> ← IEnumerable<Comic>


Book
Class
Comic
Class
→ Book
public interface Reader<T>
where T : Book
{
void Read(T item)
}
public void Demo(Reader<Book> reader)
{
Reader<Comic> comicReader = reader;
}
In C# 3 this doesn't work
Need
Have
Book ← Comic
Reader<Comic> ≠ Reader<Book>


Book
Class
Comic
Class
→ Book
public interface Reader<T>
where T : Book
{
void Read(T item)
}
public void Demo(Reader<Book> reader)
{
Reader<Comic> comicReader = reader;
}
In C# 4 this does work
Need
Contravariance
Have
Book ← Comic
Reader<Comic> → Reader<Book>

Covariance

`IEnumerable<out T>`

Uses T as an output only

Returning a subclass is always Ok.

```
public Book GetBook()
{
    return new Comic();
}
```

```
public IEnumerable<Book> GetBooks()
{
    IEnumerable<Comic> result = ...
    return result;
}
```

Contravariance

`Reader<in T>`

Uses T as an input only

Passing a subclass Is always Ok

```
public void Read(Book book) { }

Read(new Comic());
```

```
Reader<Comic> reader
    = new Reader<Book>()
reader.Read(myComic);
```

Covariance and Contravariance

Mostly, don't worry about it

It means that things
should work the way you expect.

Thanks.

twitter unrepentantgeek
email bevan@nichesoftware.co.nz
blog www.nichesoftware.co.nz