Overview

- .net core
  - Managed heap and Garbage collection
- asp.net core
  - Views, partial views
  - Passing data to views
  - Model binding

Allocating resources from the managed heap

Every program uses resources of one sort or another, be they files, memory buffers, screen space, network connections, database resources, and so on. In fact, in an object-oriented environment, every type identifies some resource available for a program's use. To use any of these resources requires memory to be allocated to represent the type. The following steps are required to access a resource:

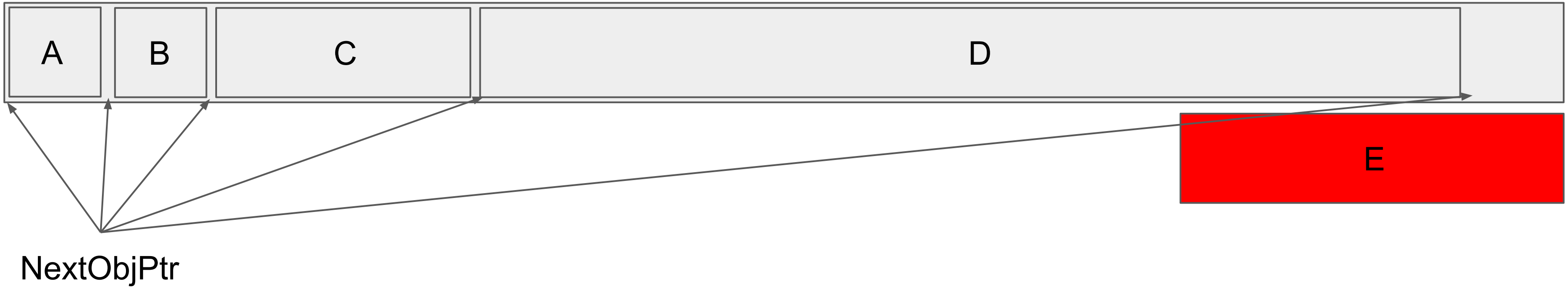
1. Allocate memory for the type that represents the resource (usually accomplished by using C#'s new operator).
2. Initialize the memory to set the initial state of the resource and to make the resource usable. The type's instance constructor is responsible for setting this initial state.
3. Use the resource by accessing the type's members (repeating as necessary).
4. Tear down the state of a resource to clean up.
5. Free the memory. The garbage collector is solely responsible for this step.

Allocating resources from the managed heap

C#'s new operator causes the CLR to perform the following steps:

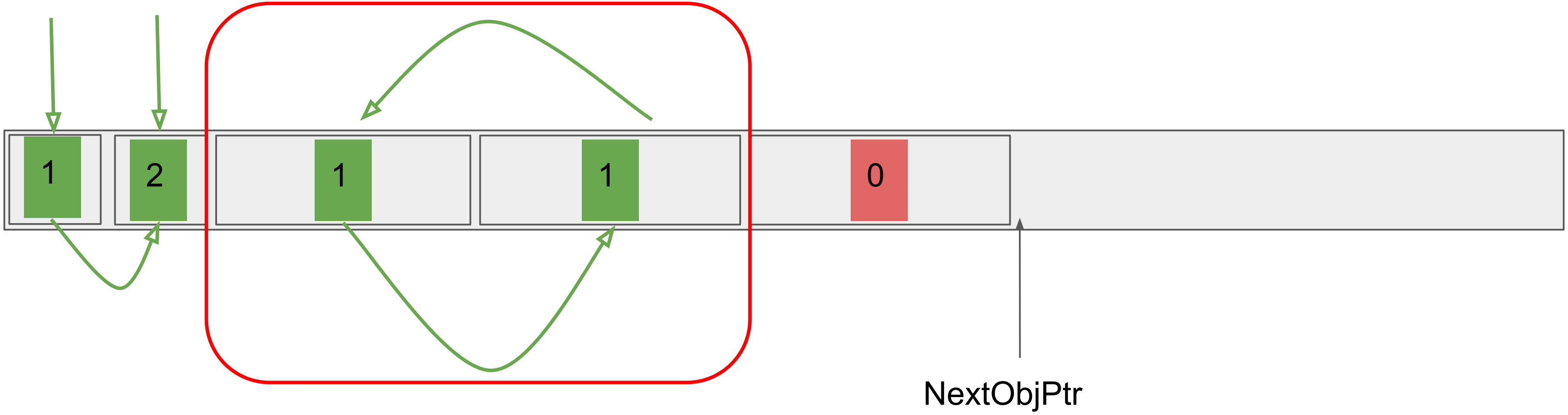
1. Calculate the number of bytes required for the type's fields (and all the fields it inherits from its base types).
2. Add the bytes required for an object's overhead. Each object has two overhead fields: a type object pointer and a sync block index. For a 32-bit application, each of these fields requires 32 bits, adding 8 bytes to each object. For a 64-bit application, each field is 64 bits, adding 16 bytes to each object.
3. The CLR then checks that the bytes required to allocate the object are available in the region. If there is enough free space in the managed heap, the object will fit, starting at the address pointed to by **NextObjPtr**, and these bytes are zeroed out. The type's constructor is called (passing **NextObjPtr** for the this parameter), and the new operator returns a reference to the object. Just before the reference is returned, **NextObjPtr** is advanced past the object and now points to the address where the next object will be placed in the heap.

Allocating resources from the managed heap



Reference counting

For managing the lifetime of objects, some systems use a reference counting algorithm. With a reference counting system, each object on the heap maintains an internal field indicating how many “parts” of the program are currently using that object. As each “part” gets to a place in the code where it no longer requires access to an object, it decrements that object’s count field. When the count field reaches 0, the object deletes itself from memory.



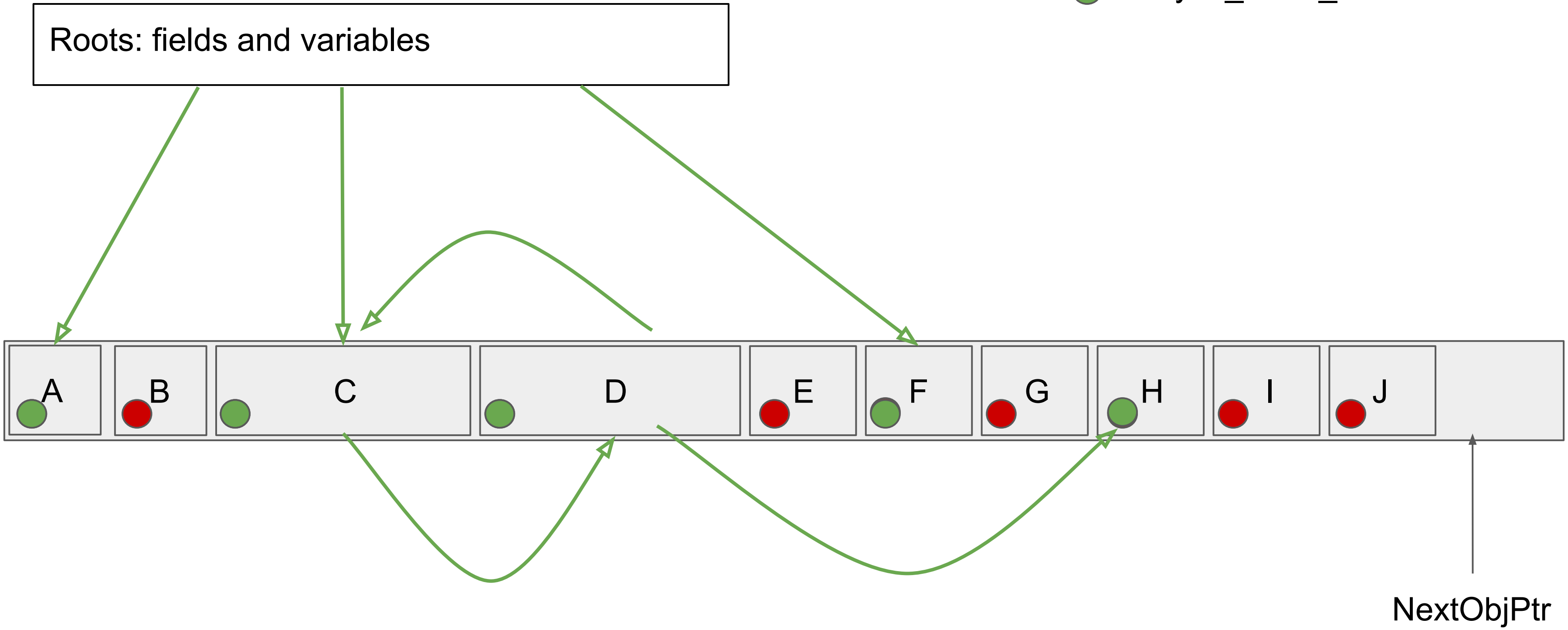
Referencing tracking

Due to the problem with reference counting garbage collector algorithms, CLR uses a referencing tracking algorithm instead. The reference tracking algorithm cares only about reference type variables, because only these variables can refer to an object on the heap; value type variables contain the value type instance directly. Reference type variables can be used in many contexts: static and instance fields within a class or a method’s arguments or local variables. We refer to all reference type variables as roots.

Marking phase

🔴 Sync_block_index = 0

🟢 Sync_block_index = 1



Roots: fields and variables

A

B

C

D

E

F

G

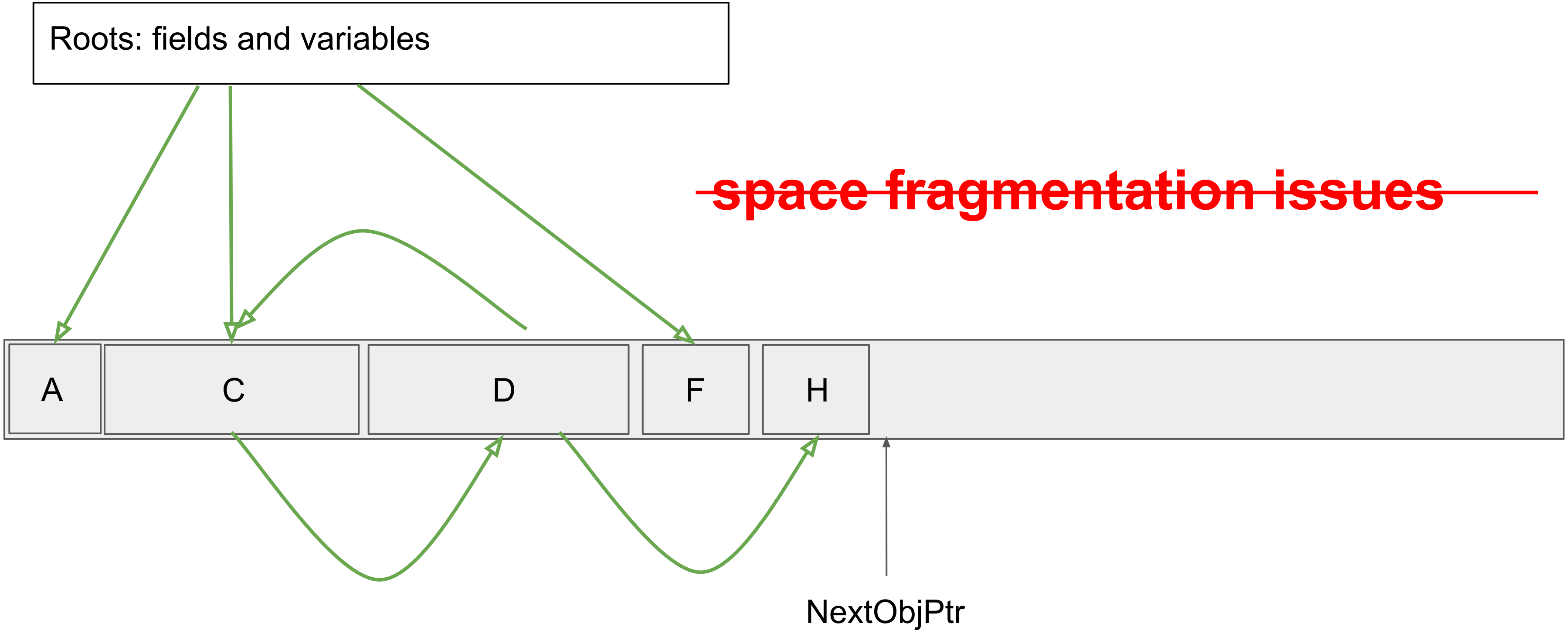
H

I

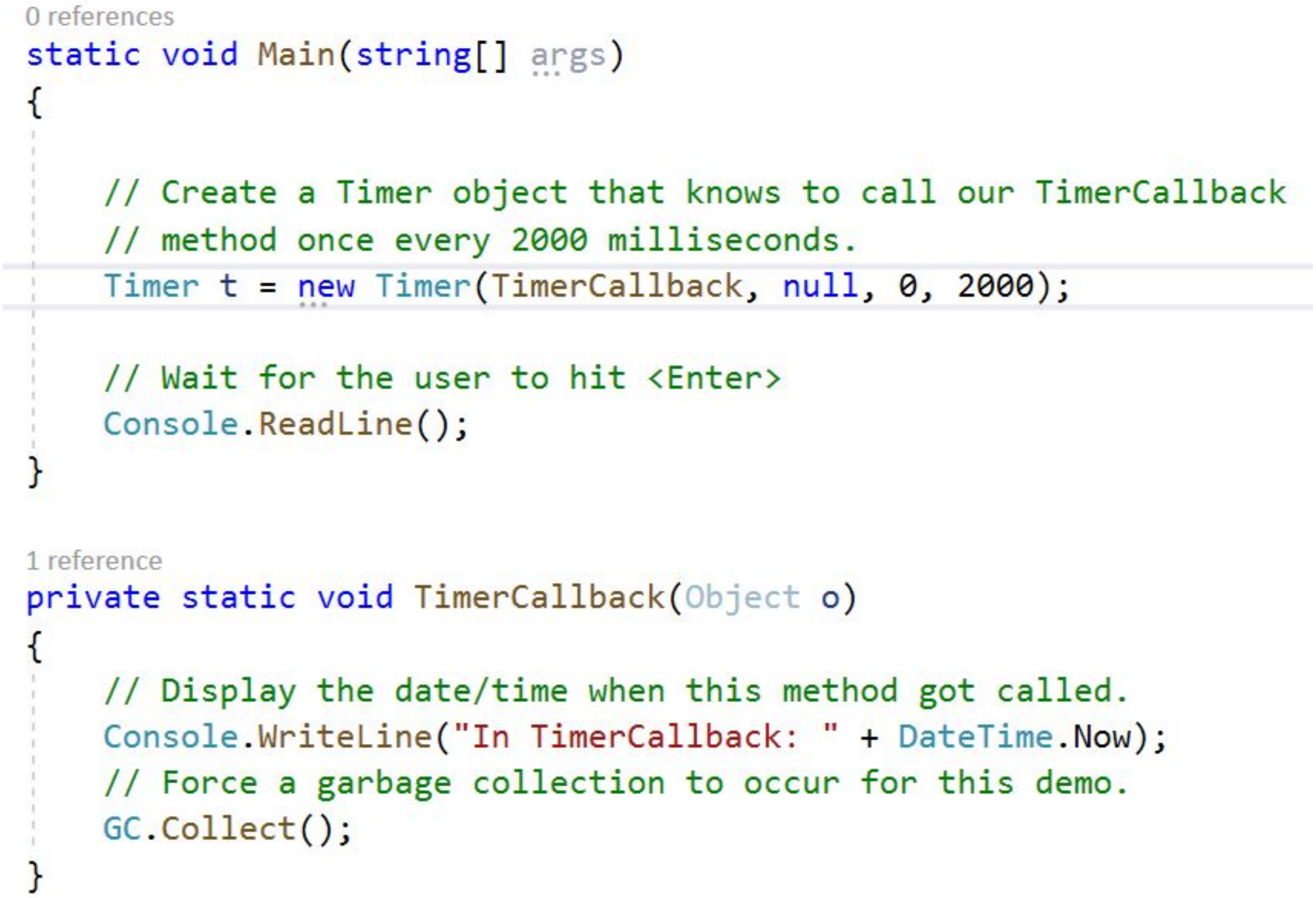
J

NextObjPtr

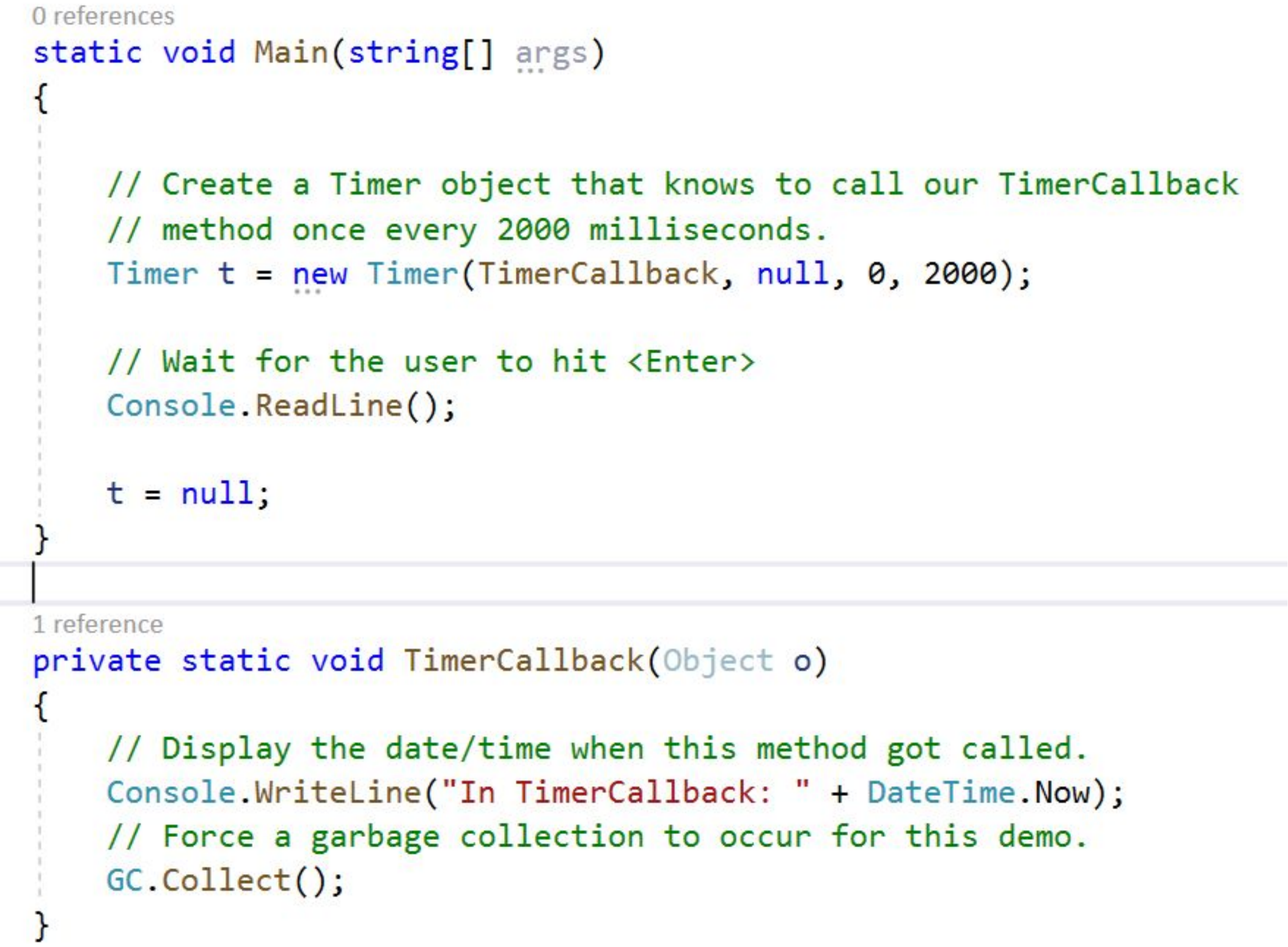
Compacting phase



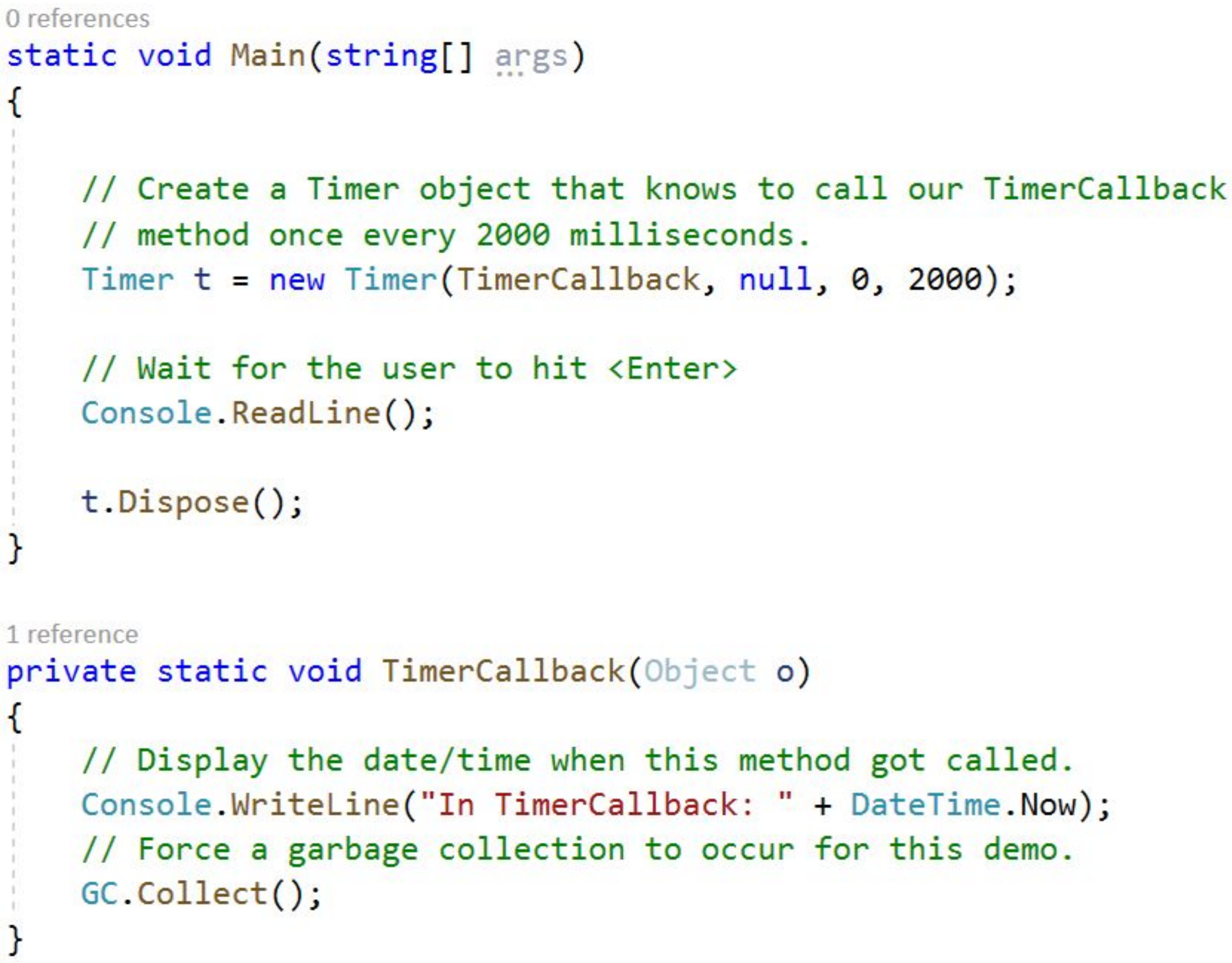
~~space fragmentation issues~~



```
0 references
static void Main(string[] args)
{

    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>
    Console.ReadLine();
}

1 reference
private static void TimerCallback(Object o)
{
    // Display the date/time when this method got called.
    Console.WriteLine("In TimerCallback: " + DateTime.Now);
    // Force a garbage collection to occur for this demo.
    GC.Collect();
}
```



```
0 references
static void Main(string[] args)
{

    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>
    Console.ReadLine();

    t = null;
}

1 reference
private static void TimerCallback(Object o)
{
    // Display the date/time when this method got called.
    Console.WriteLine("In TimerCallback: " + DateTime.Now);
    // Force a garbage collection to occur for this demo.
    GC.Collect();
}
```

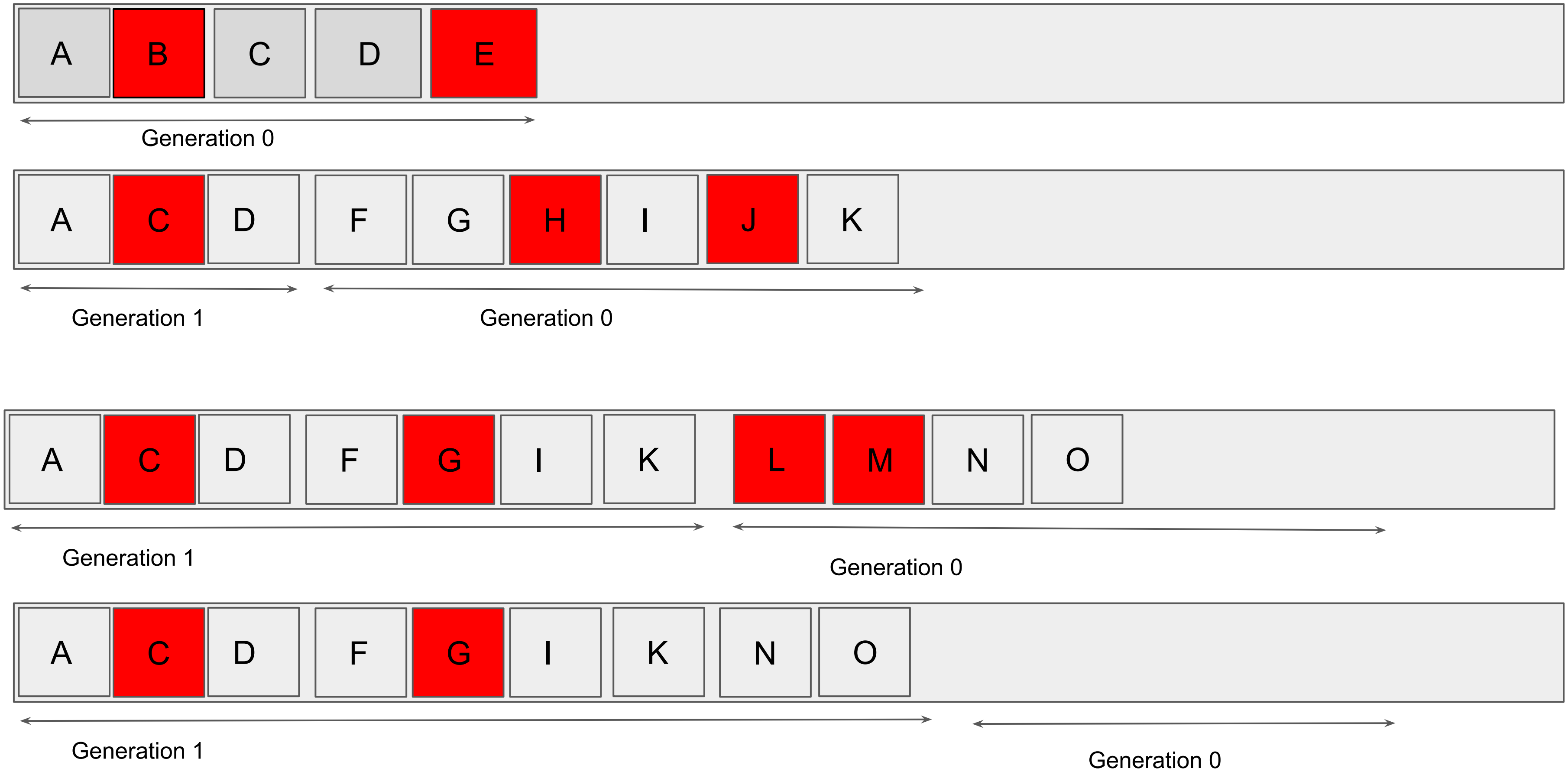


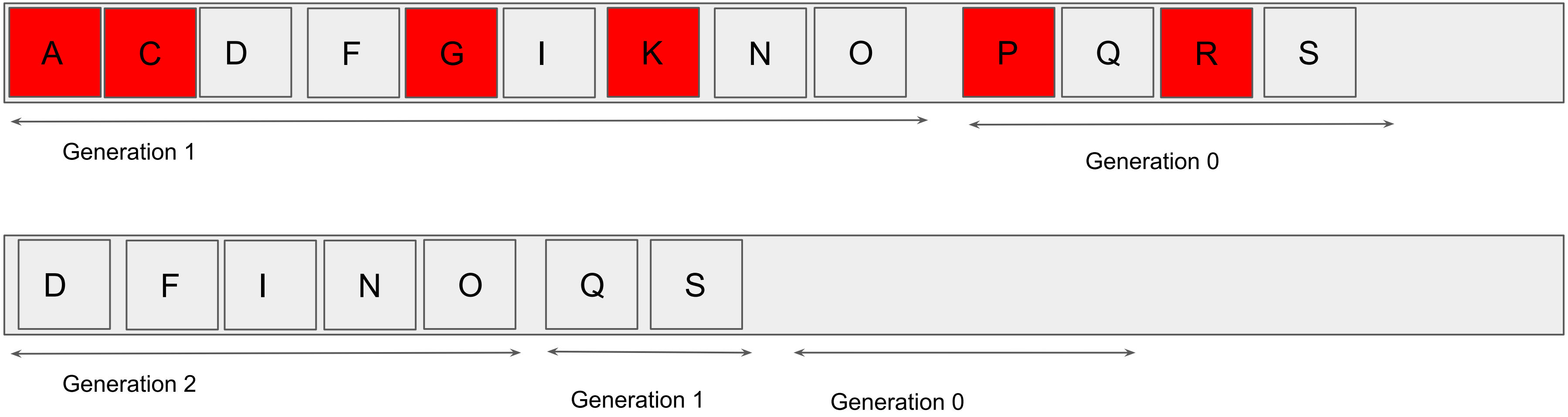
```
0 references
static void Main(string[] args)
{

    // Create a Timer object that knows to call our TimerCallback
    // method once every 2000 milliseconds.
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Wait for the user to hit <Enter>
    Console.ReadLine();

    t.Dispose();
}

1 reference
private static void TimerCallback(Object o)
{
    // Display the date/time when this method got called.
    Console.WriteLine("In TimerCallback: " + DateTime.Now);
    // Force a garbage collection to occur for this demo.
    GC.Collect();
}
```

Generations: Improving Performance

The CLR’s GC is a generational garbage collector (also known as an ephemeral garbage collector, although I don’t use the latter term in this book). A generational GC makes the following assumptions about your code:

- The newer an object is, the shorter its lifetime will be.
- The older an object is, the longer its lifetime will be.
- Collecting a portion of the heap is faster than collecting the whole heap.


A
B
C
D
E
Generation 0
A
C
D
F
G
H
I
J
K
Generation 1
Generation 0
A
C
D
F
G
I
K
L
M
N
O
Generation 1
Generation 0
A
C
D
F
G
I
K
N
O
Generation 1
Generation 0



A C D F G I K N O P Q R S

Generation 1

Generation 0

D F I N O Q S

Generation 2

Generation 1

Generation 0

Large Objects

The CLR considers each single object to be either a small object or a large object. So far, in this chapter, I’ve been focusing on small objects. Today, a large object is 85,000 bytes or more in size.24 The CLR treats large objects slightly differently that how it treats small objects:

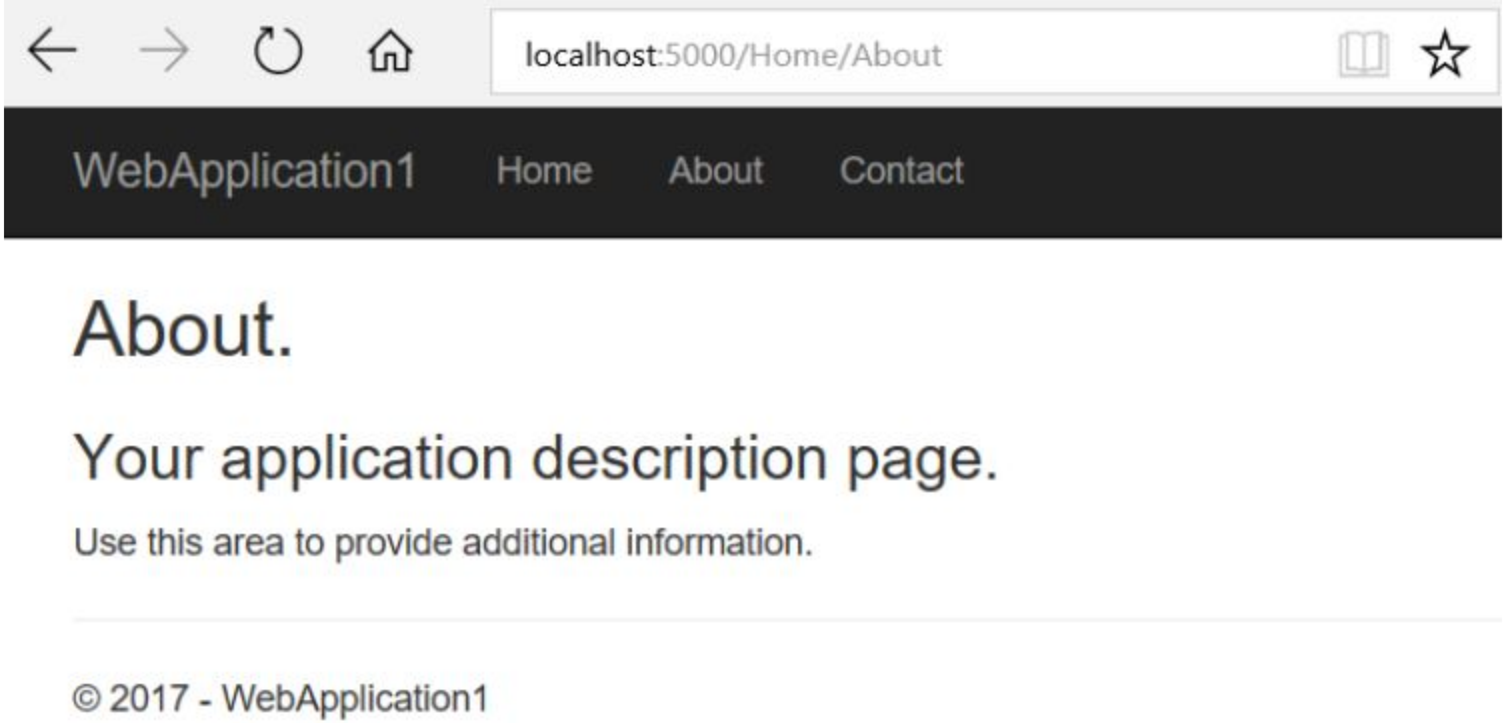
- Large objects are not allocated within the same address space as small objects; they are allocated elsewhere within the process’ address space.

- Today, the GC doesn’t compact large objects because of the time it would require to move them in memory. For this reason, address space fragmentation can occur between large objects within the process leading to an OutOfMemoryException being thrown. In a future version of the CLR, large objects may participate in compaction.

- Large objects are immediately considered to be part of generation 2; they are never in generation 0 or 1. So, you should create large objects only for resources that you need to keep alive for a long time. Allocating short-lived large objects will cause generation 2 to be collected more frequently, hurting performance. Usually large objects are large strings (like XML or JSON) or byte arrays which you use for I/O operations, such as reading bytes from a file or network

Asp.net core mvc

Views



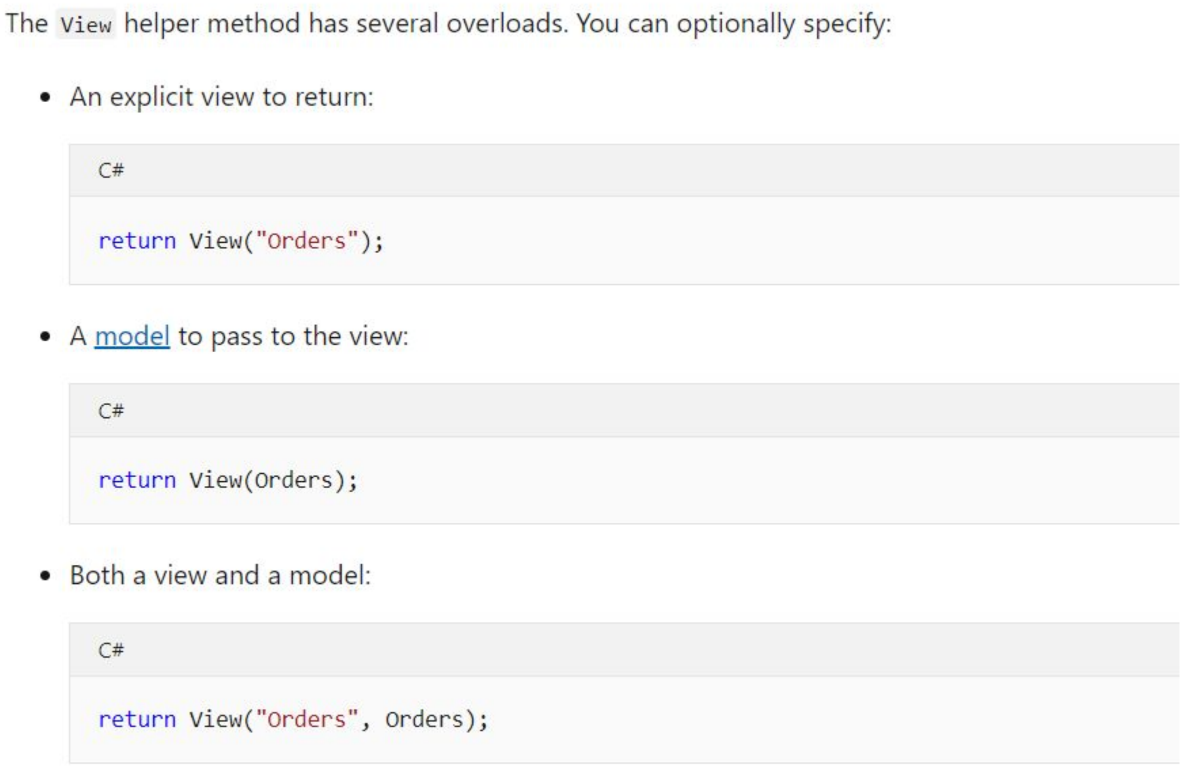
About.

Your application description page.

Use this area to provide additional information.

© 2017 - WebApplication1

Returning view



The `View` helper method has several overloads. You can optionally specify:

- An explicit view to return:

```
C#

return View("Orders");
```

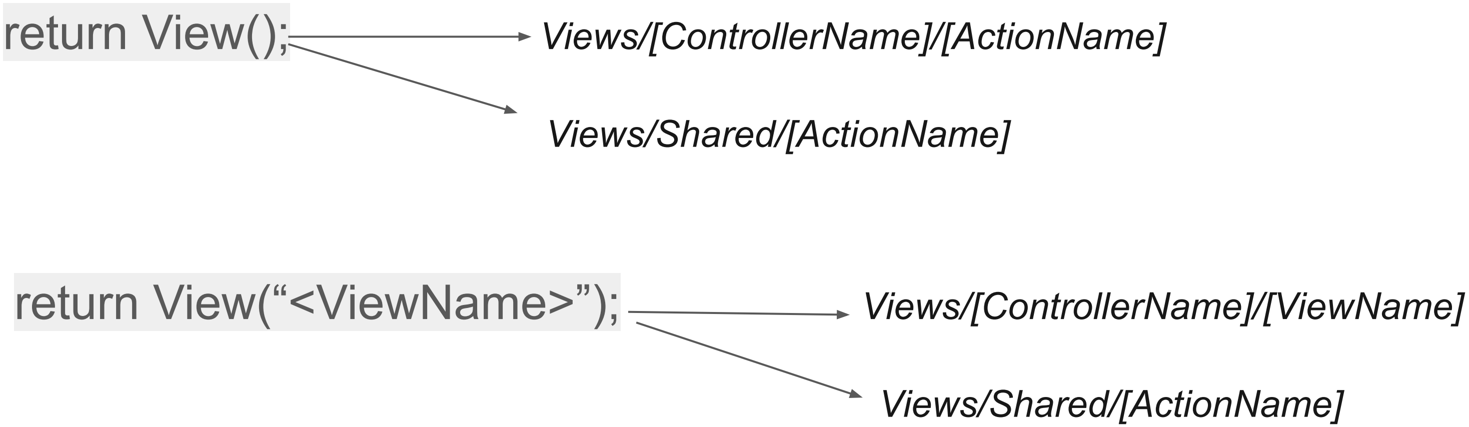
- A model to pass to the view:

```
C#

return View(Orders);
```

- Both a view and a model:

```
C#

return View("Orders", Orders);
```

View discovery



`return View();`

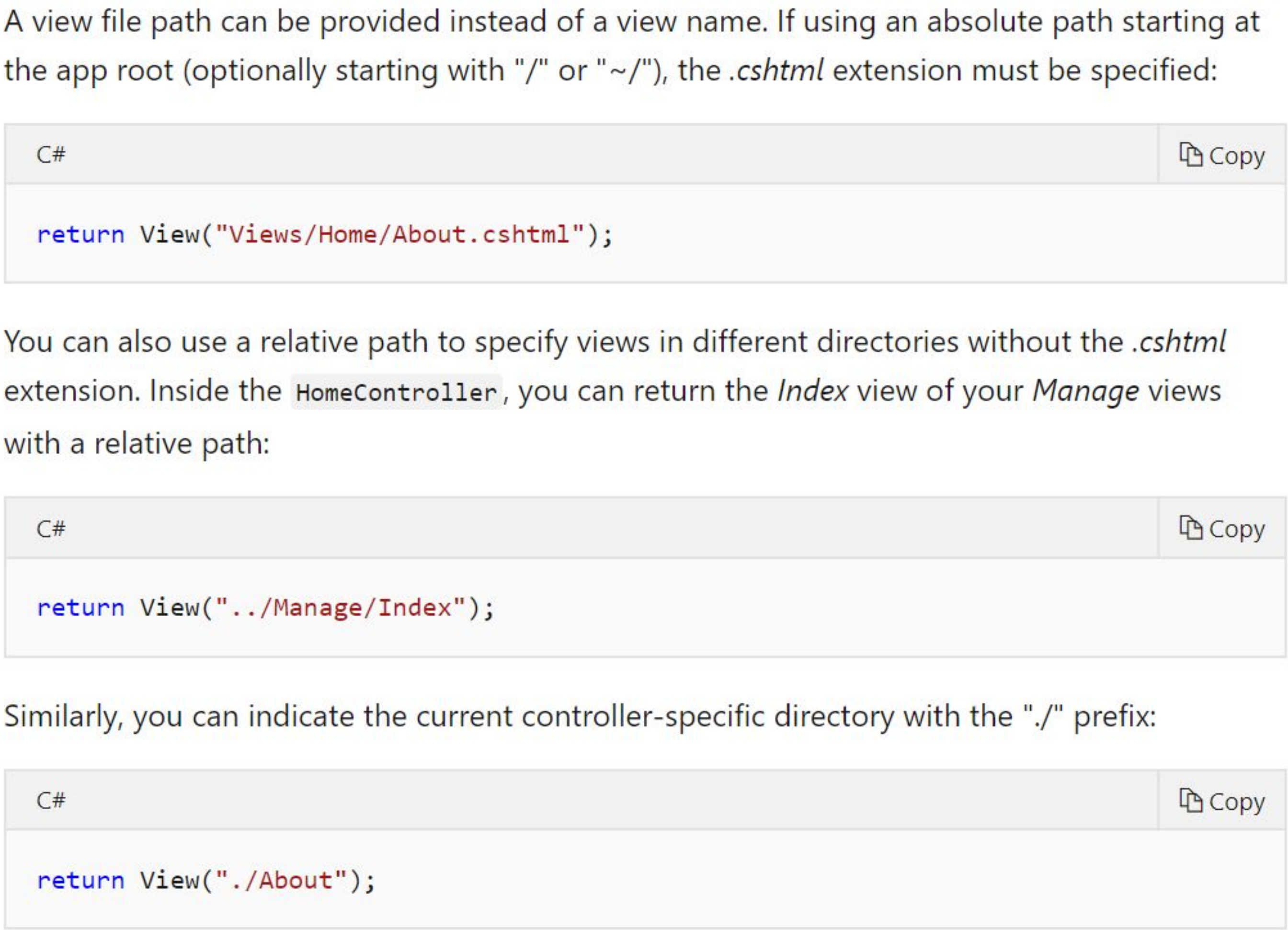
Views/[ControllerName]/[ActionName]

Views/Shared/[ActionName]

`return View("<ViewName>");`

Views/[ControllerName]/[ViewName]

Views/Shared/[ActionName]



A view file path can be provided instead of a view name. If using an absolute path starting at the app root (optionally starting with "/" or "~/"), the *.cshtml* extension must be specified:

```C#
return View("Views/Home/About.cshtml");
```

You can also use a relative path to specify views in different directories without the *.cshtml* extension. Inside the `HomeController`, you can return the *Index* view of your *Manage* views with a relative path:

```C#
return View("../Manage/Index");
```

Similarly, you can indicate the current controller-specific directory with the "./" prefix:

```C#
return View("./About");
```

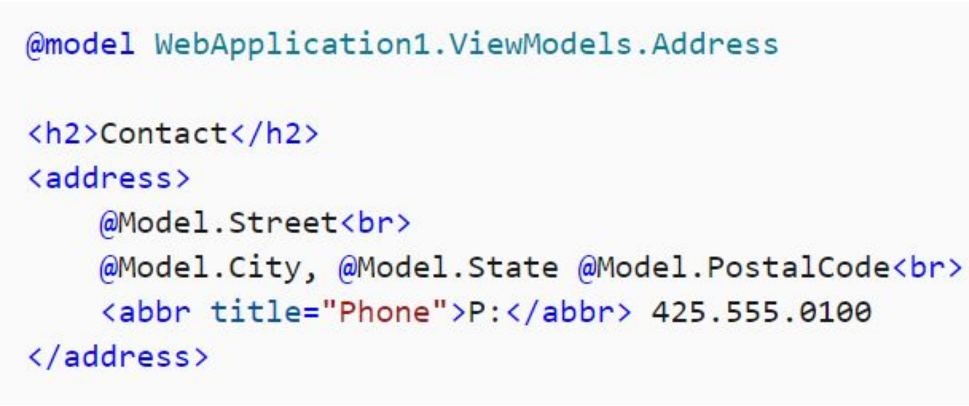
Passing data to view

Pass data to views using several approaches:

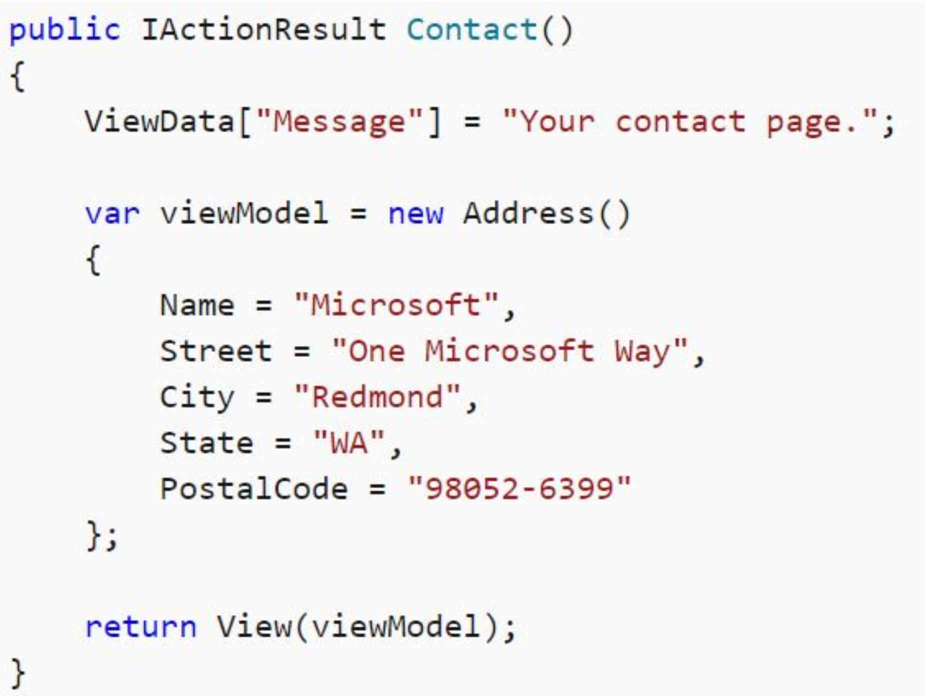
- Strongly typed data: viewmodel
- Weakly typed data
  - `ViewData` (`ViewDataAttribute`)
  - `ViewBag`

Strongly typed view model

The most robust approach is to specify a model type in the view. This model is commonly referred to as a *viewmodel*. You pass an instance of the viewmodel type to the view from the action.

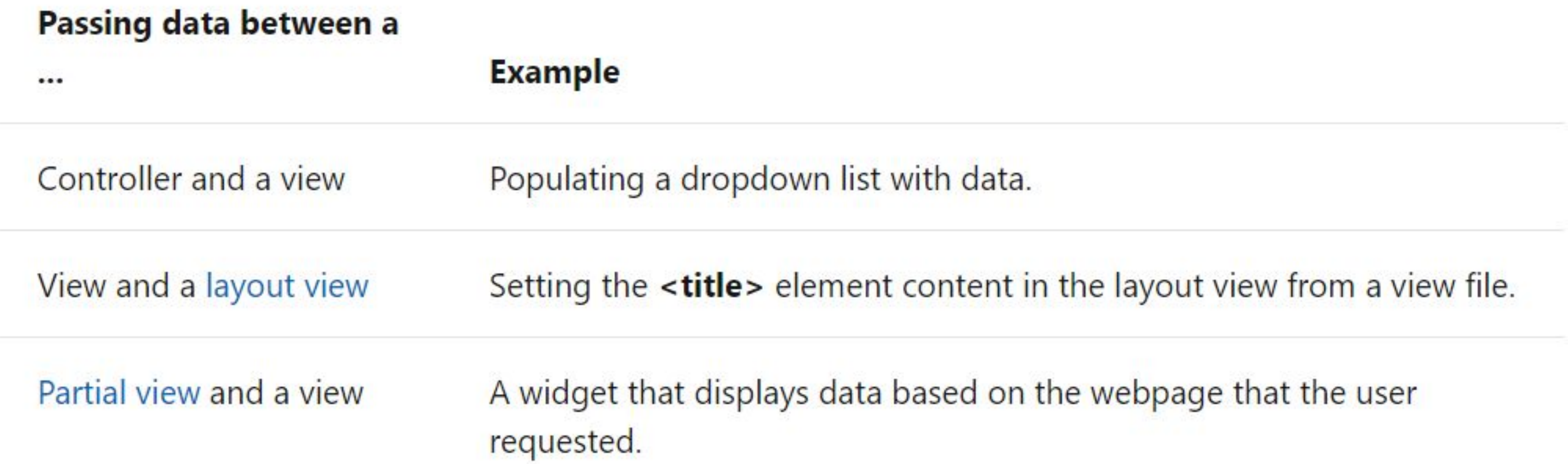


```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```



```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

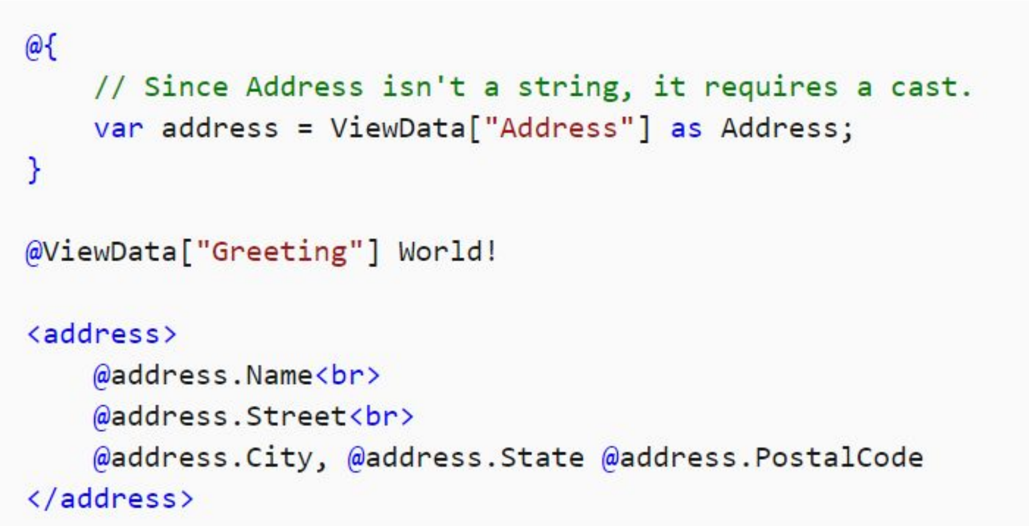
Weakly typed data (ViewData,ViewBag)



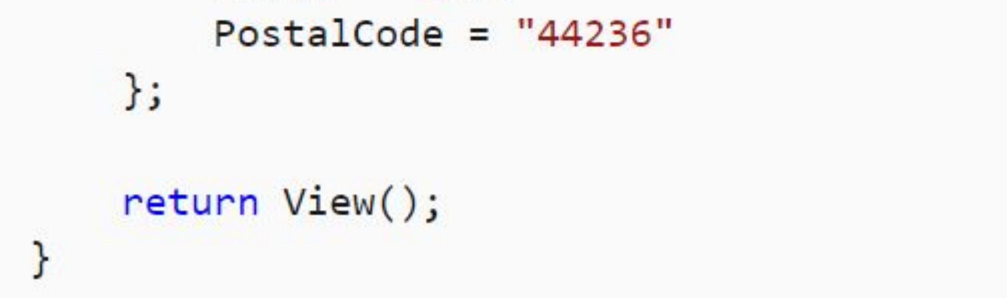
Passing data between a ...	Example
Controller and a view	Populating a dropdown list with data.
View and a layout view	Setting the **<title>** element content in the layout view from a view file.
Partial view and a view	A widget that displays data based on the webpage that the user requested.

View data

ViewData is a ViewDataDictionary object accessed through string keys. String data can be stored and used directly without the need for a cast, but you must cast other ViewData object values to specific types when you extract them. You can use ViewData to pass data from controllers to views and within views, including partial views and layouts.

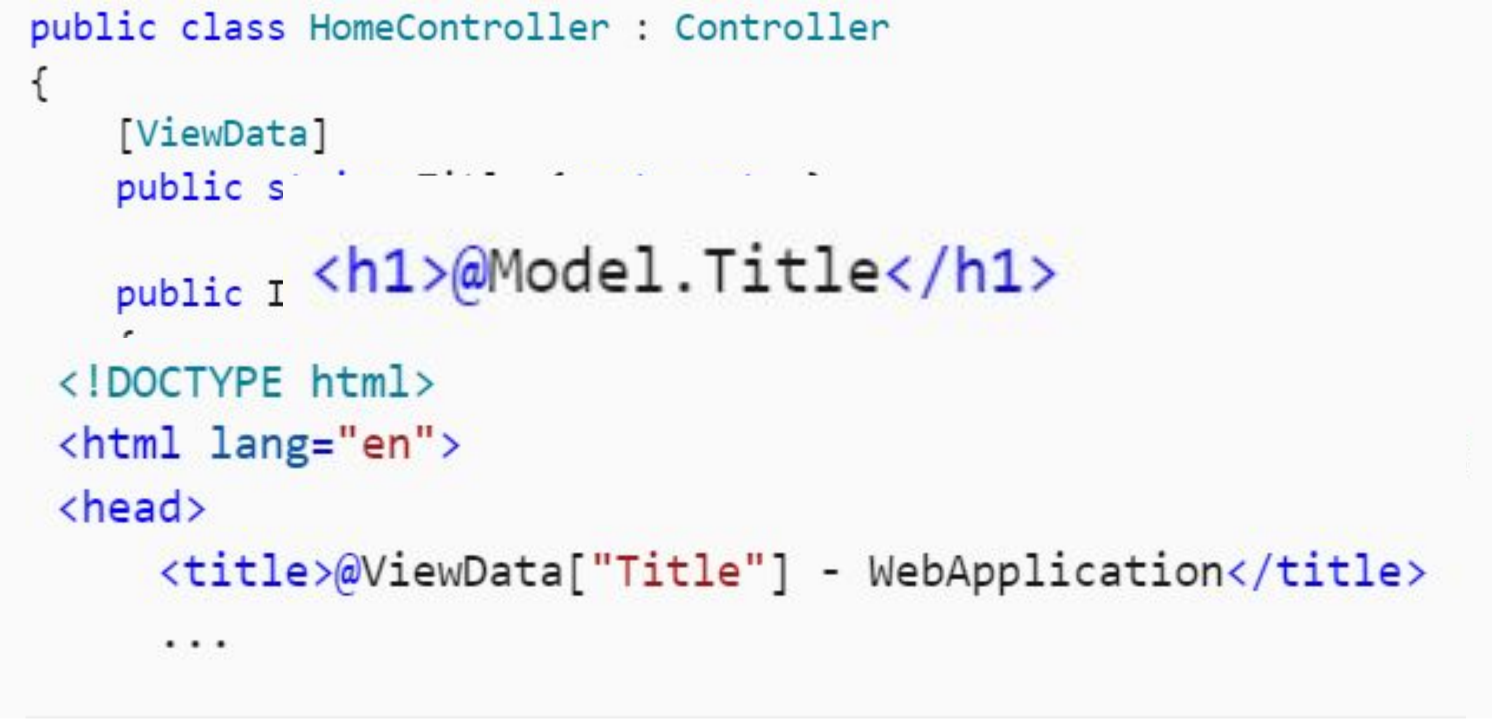


```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```



```
            PostalCode = "44236"
     };

     return View();
}
```

ViewData attribute



```
public class HomeController : Controller
{
    [ViewData]
    public s

    public I
```

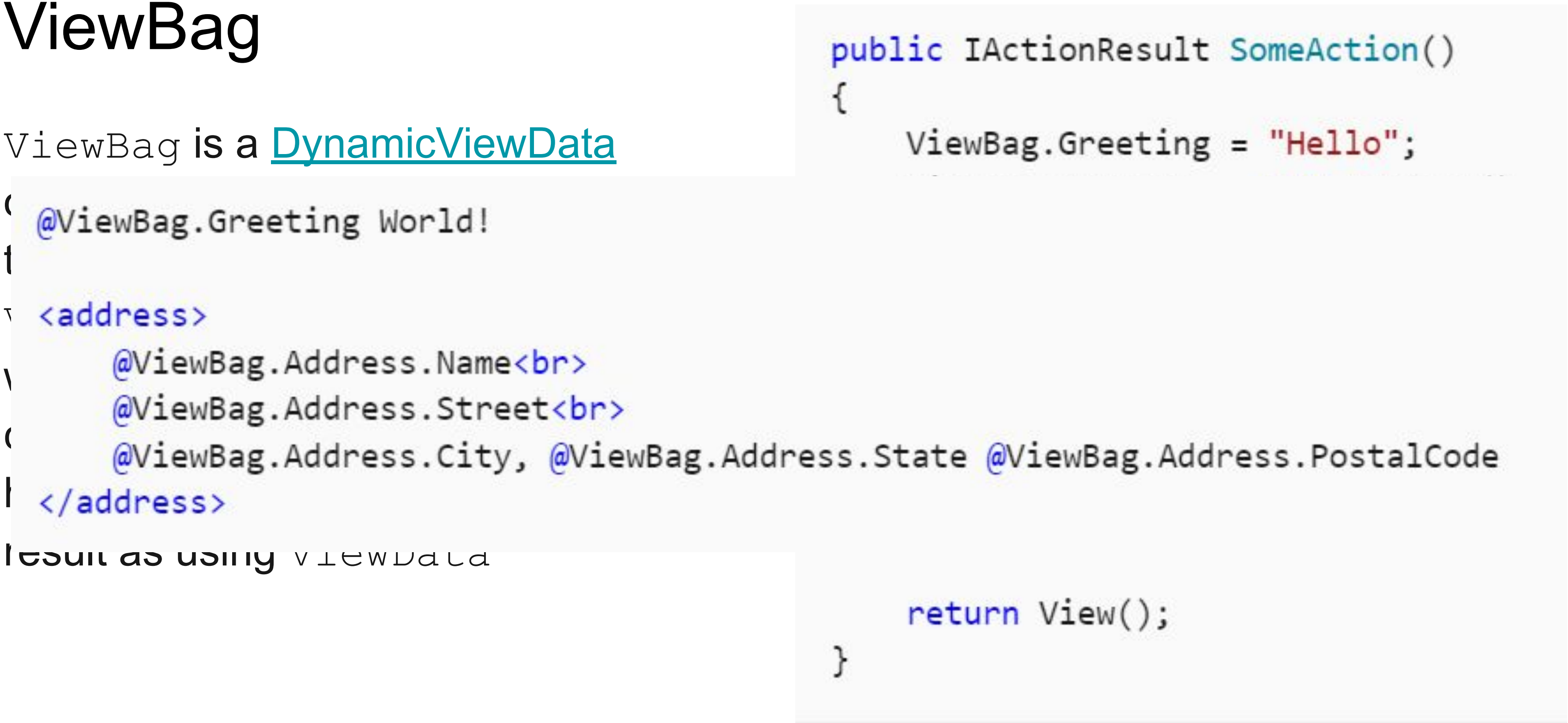
```
<h1>@Model.Title</h1>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

ViewBag

ViewBag is a DynamicViewData

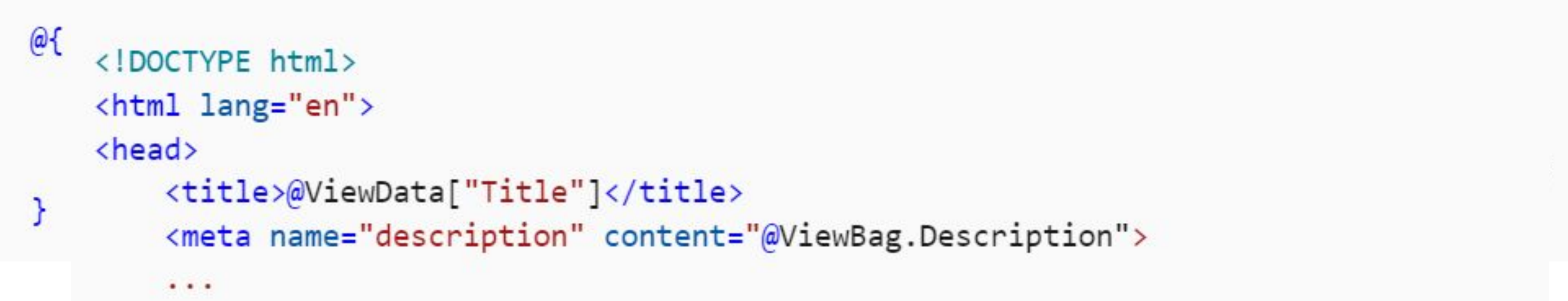
result as using ViewData



```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";



    return View();
}
```

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>
```

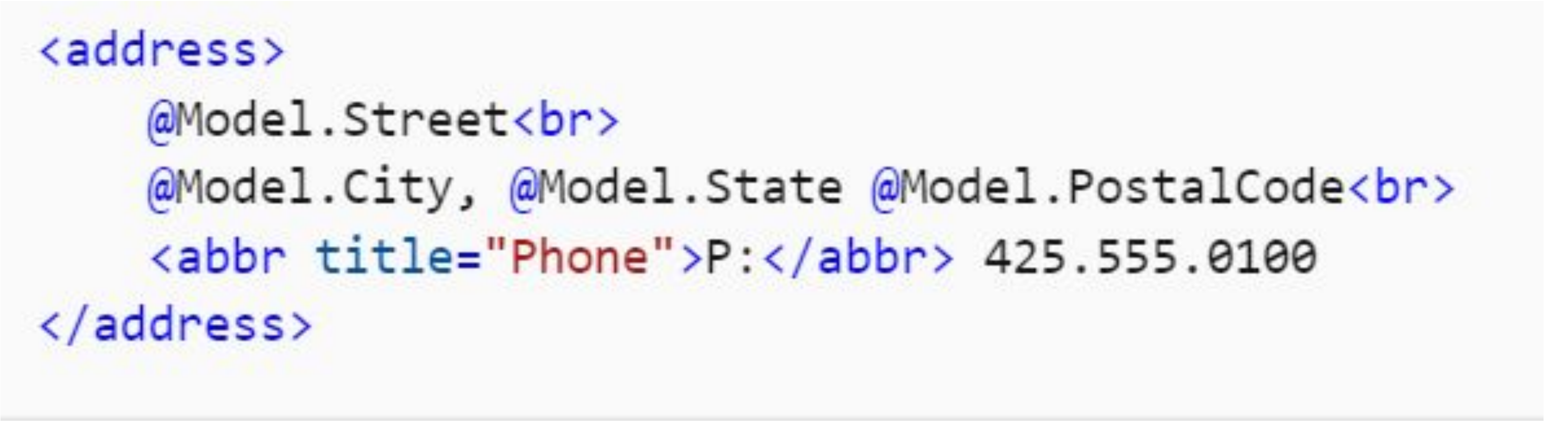
Using ViewBag and ViewData simultaneously



```
@{
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <title>@ViewData["Title"]</title>
}
        <meta name="description" content="@ViewBag.Description">
        ...
```

- `ViewData`
  - Derives from ViewDataDictionary, so it has dictionary properties that can be useful, such as `ContainsKey`, `Add`, `Remove`, and `Clear`.
  - Keys in the dictionary are strings, so whitespace is allowed. Example: `ViewData["Some Key With Whitespace"]`
  - Any type other than a `string` must be cast in the view to use `ViewData`.
- `ViewBag`
  - Derives from DynamicViewData, so it allows the creation of dynamic properties using dot notation (`@ViewBag.SomeKey = <value or object>`), and no casting is required. The syntax of `ViewBag` makes it quicker to add to controllers and views.
  - Simpler to check for null values. Example: `@ViewBag.Person?.Name`

Dynamic views



```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

Partial views in asp.net core

A partial view is a Razor markup file (*.cshtml*) that renders HTML output *within* another markup file's rendered output.

When to use

Partial views are an effective way to:

- Break up large markup files into smaller components.
  In a large, complex markup file composed of several logical pieces, there's an advantage to working with each piece isolated into a partial view. The code in the markup file is manageable because the markup only contains the overall page structure and references to partial views.
- Reduce the duplication of common markup content across markup files.
  When the same markup elements are used across markup files, a partial view removes the duplication of markup content into one partial view file. When the markup is changed in the partial view, it updates the rendered output of the markup files that use the partial view.

Partial views shouldn't be used to maintain common layout elements. Common layout elements should be specified in _Layout.cshtml files.

Don't use a partial view where complex rendering logic or code execution is required to render the markup. Instead of a partial view, use a view component.

Declare partial views

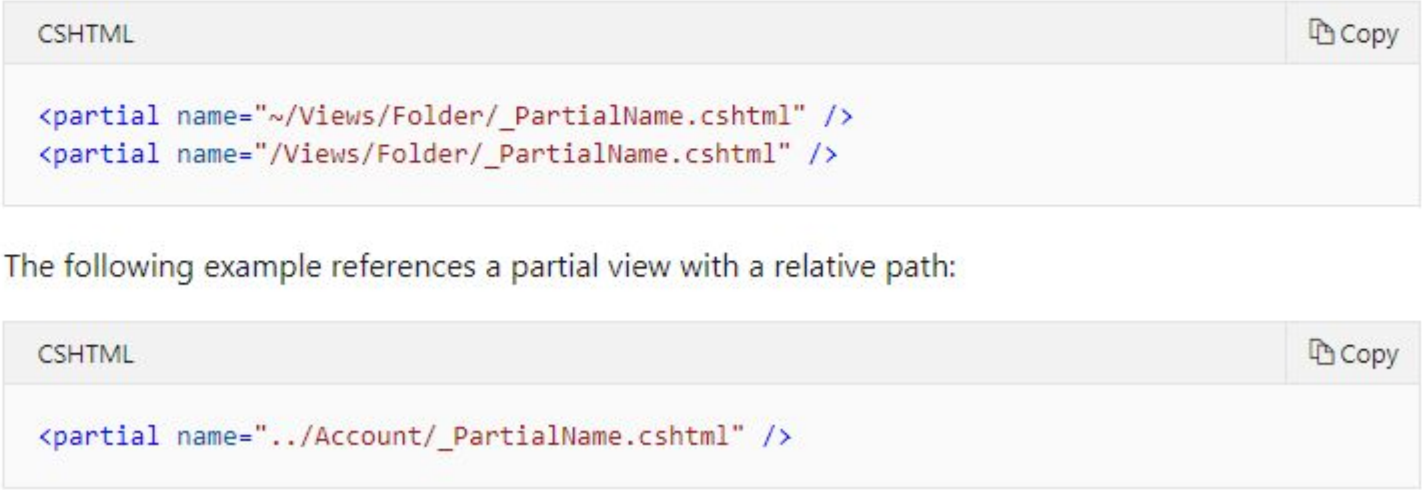
A partial view is a *.cshtml* markup file maintained within the *Views* folder (MVC)

In ASP.NET Core MVC, a controller's ViewResult is capable of returning either a view or a partial view

Unlike MVC view or page rendering, a partial view doesn't run *_ViewStart.cshtml*. For more information on *_ViewStart.cshtml*

Partial view file names often begin with an underscore (_). This naming convention isn't required, but it helps to visually differentiate partial views from views and pages.

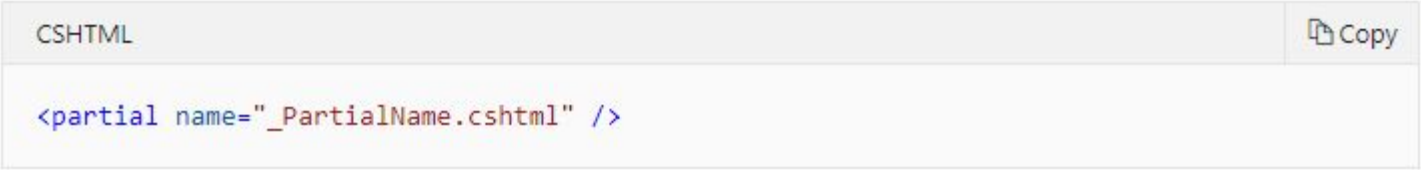
Reference partial view



```
<partial name="~/Views/Folder/_PartialName.cshtml" />
<partial name="/Views/Folder/_PartialName.cshtml" />
```

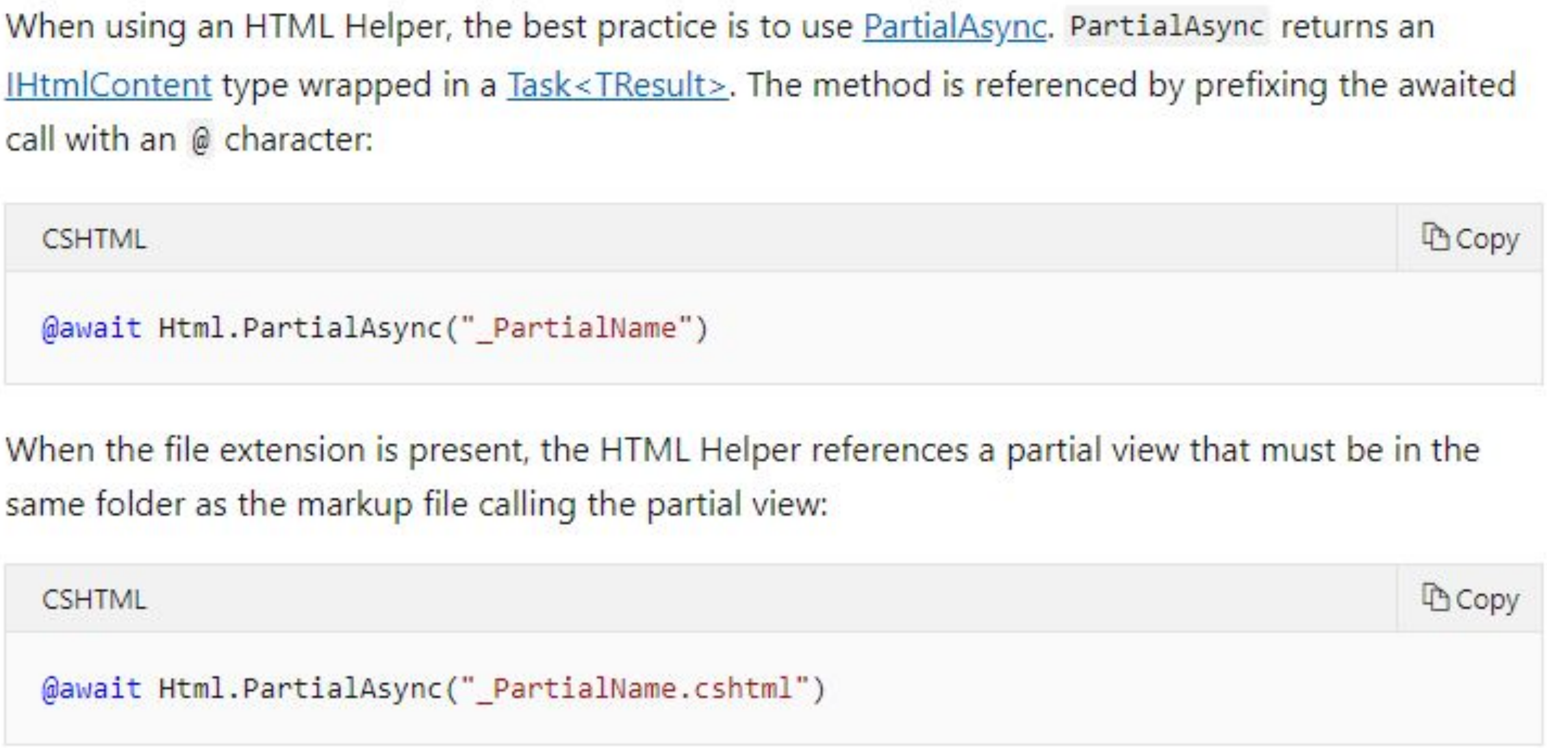
The following example references a partial view with a relative path:

```
<partial name="../Account/_PartialName.cshtml" />
```



```
<partial name="_PartialName.cshtml" />
```

Asynchronous html helpers

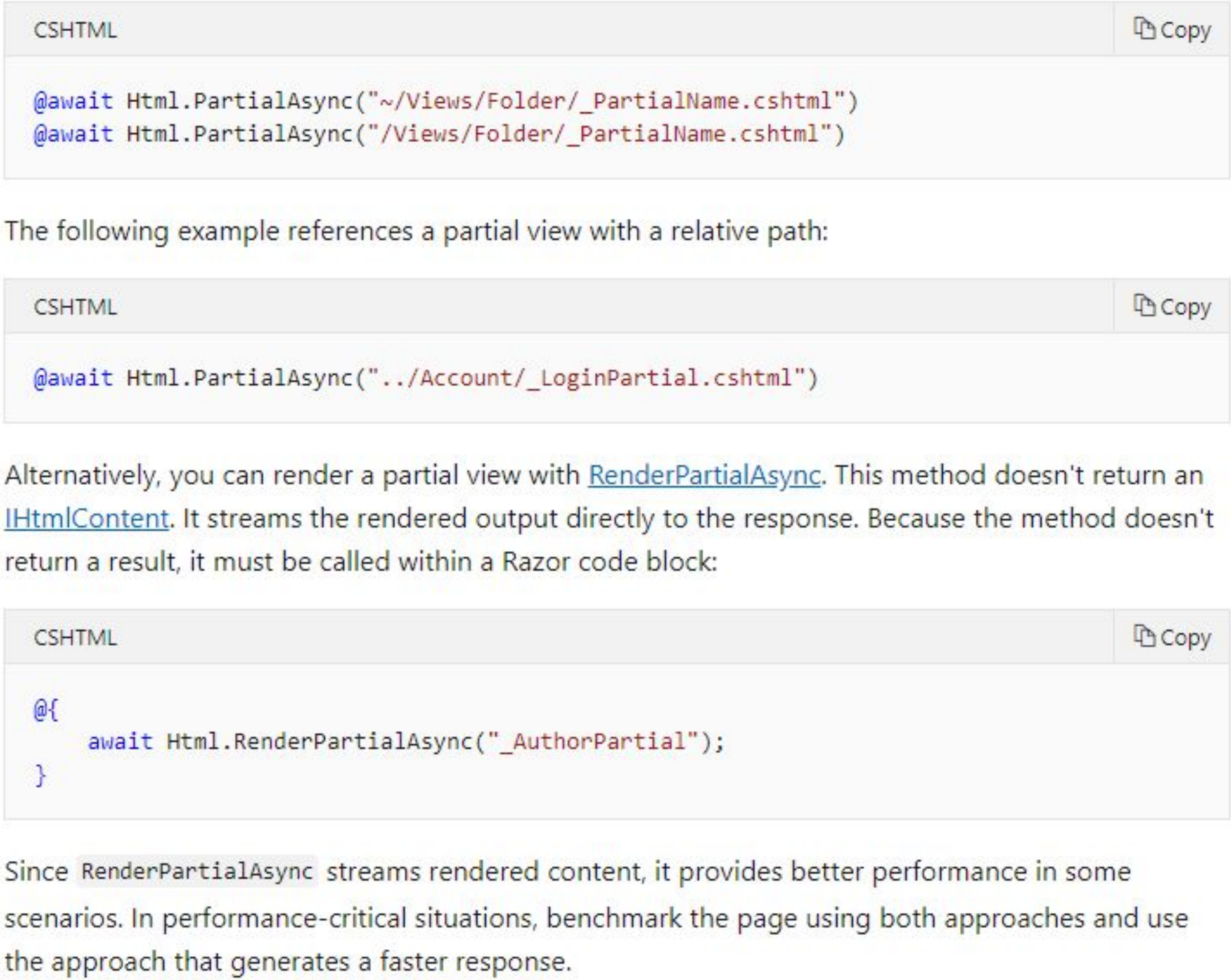


When using an HTML Helper, the best practice is to use PartialAsync. `PartialAsync` returns an IHtmlContent type wrapped in a Task<TResult>. The method is referenced by prefixing the awaited call with an `@` character:

```cshtml
@await Html.PartialAsync("_PartialName")
```

When the file extension is present, the HTML Helper references a partial view that must be in the same folder as the markup file calling the partial view:

```cshtml
@await Html.PartialAsync("_PartialName.cshtml")
```



CSHTML Copy

```cshtml
@await Html.PartialAsync("~/Views/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Views/Folder/_PartialName.cshtml")
```

The following example references a partial view with a relative path:

CSHTML Copy

```cshtml
@await Html.PartialAsync("../Account/_LoginPartial.cshtml")
```

Alternatively, you can render a partial view with RenderPartialAsync. This method doesn't return an IHtmlContent. It streams the rendered output directly to the response. Because the method doesn't return a result, it must be called within a Razor code block:

CSHTML Copy

```cshtml
@{
    await Html.RenderPartialAsync("_AuthorPartial");
}
```

Since `RenderPartialAsync` streams rendered content, it provides better performance in some scenarios. In performance-critical situations, benchmark the page using both approaches and use the approach that generates a faster response.

Partial view discovery

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`
4. `/Pages/Shared`

Partial view discovery

The following conventions apply to partial view discovery:

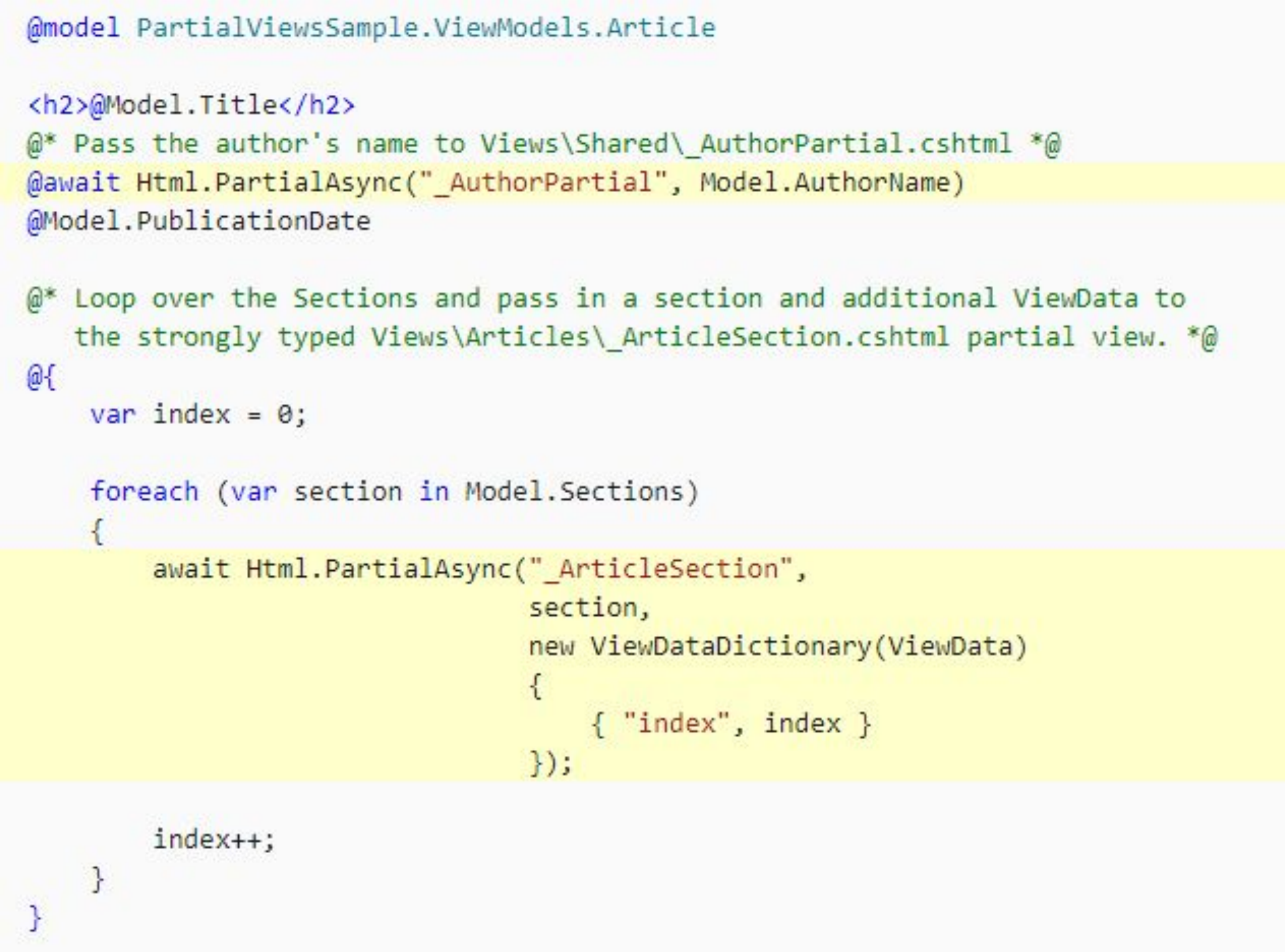
- Different partial views with the same file name are allowed when the partial views are in different folders.
- When referencing a partial view by name without a file extension and the partial view is present in both the caller's folder and the *Shared* folder, the partial view in the caller's folder supplies the partial view. If the partial view isn't present in the caller's folder, the partial view is provided from the *Shared* folder. Partial views in the *Shared* folder are called *shared partial views* or *default partial views*.
- Partial views can be *chained*—a partial view can call another partial view if a circular reference isn't formed by the calls. Relative paths are always relative to the current file, not to the root or parent of the file.

Access data from partial view

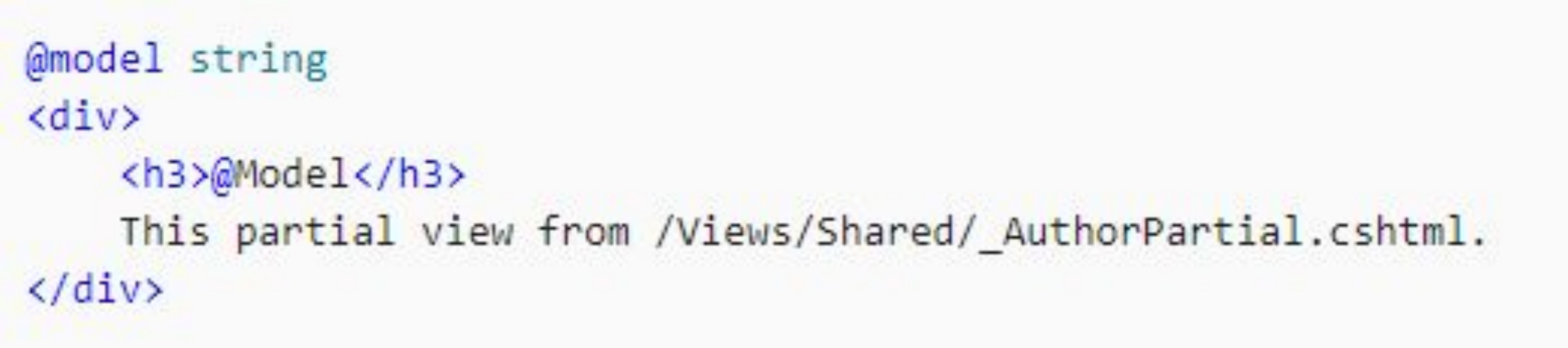
When a partial view is instantiated, it receives a *copy* of the parent's `ViewData` dictionary. Updates made to the data within the partial view aren't persisted to the parent view. `ViewData` changes in a partial view are lost when the partial view returns.

You can pass a model into a partial view. The model can be a custom object. You can pass a model with `PartialAsync` (renders a block of content to the caller) or `RenderPartialAsync` (streams the content to the output):

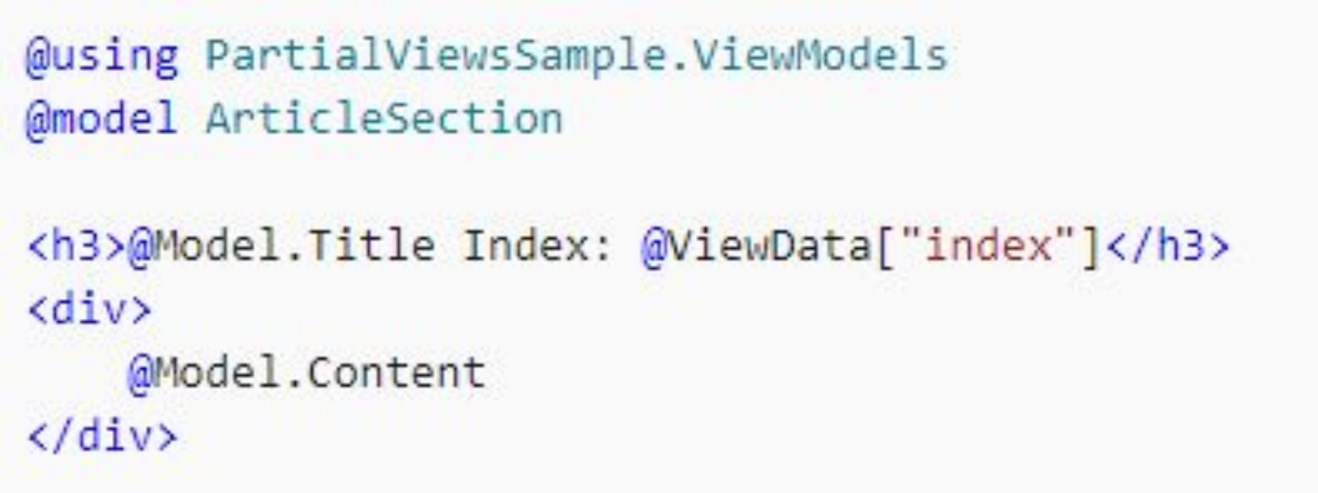
```
@await Html.PartialAsync("_PartialName", model)
```



```
@model PartialViewsSample.ViewModels.Article

<h2>@Model.Title</h2>
@* Pass the author's name to Views\Shared\_AuthorPartial.cshtml *@
@await Html.PartialAsync("_AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Views\Articles\_ArticleSection.cshtml partial view. *@
@{
    var index = 0;

    foreach (var section in Model.Sections)
    {
        await Html.PartialAsync("_ArticleSection",
                                section,
                                new ViewDataDictionary(ViewData)
                                {
                                    { "index", index }
                                });

        index++;
    }
}
```



```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Views/Shared/_AuthorPartial.cshtml.
</div>
```



```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

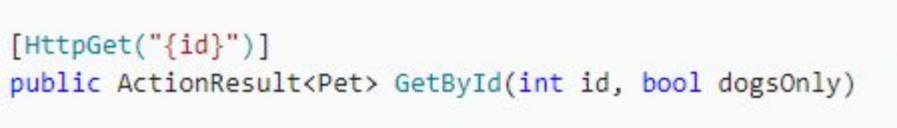
Model binding

What is model binding

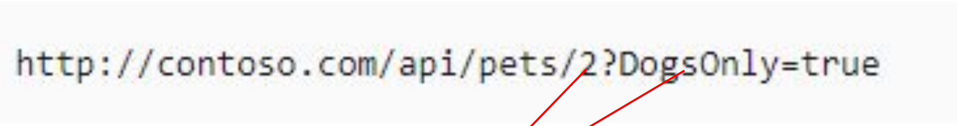
Controllers works with data that comes from HTTP requests. For example, route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

- Retrieves data from various sources such as route data, form fields, and query strings.
- Provides the data to controllers in method parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.

Example



```
[HttpGet("{id}")]
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```



```
http://contoso.com/api/pets/2?DogsOnly=true
```

Model binding goes though the following steps after the routing system selects the action method:

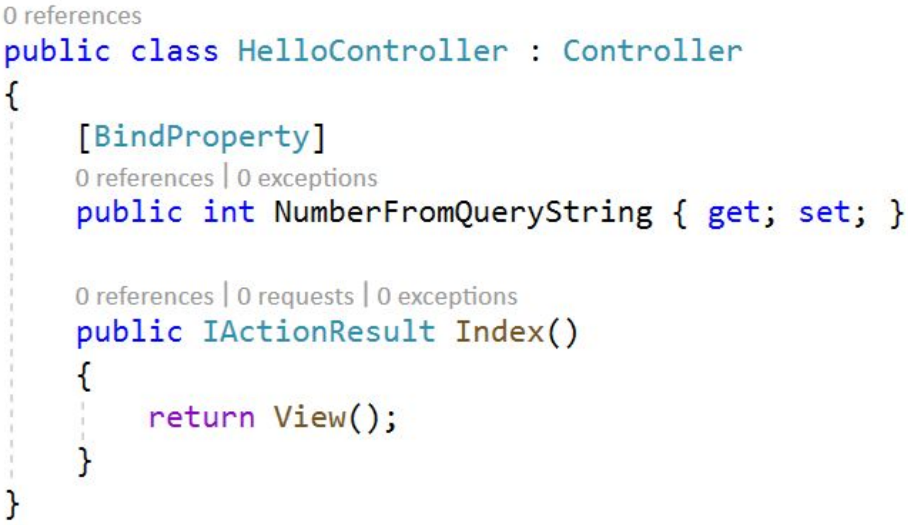
- Finds the first parameter of `GetByID`, an integer named `id`.
- Looks through the available sources in the HTTP request and finds `id` = "2" in route data.
- Converts the string "2" into integer 2.
- Finds the next parameter of `GetByID`, a boolean named `dogsOnly`.
- Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
- Converts the string "true" into boolean `true`.

Targets

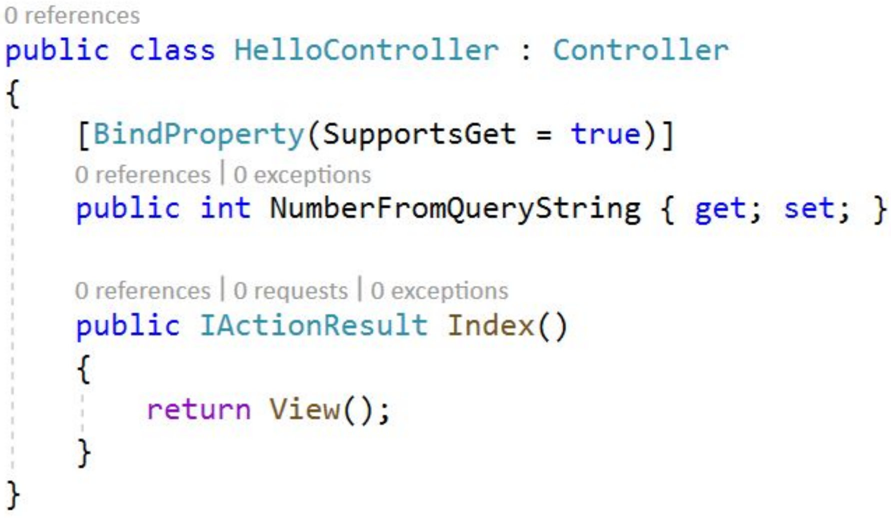
Model binding tries to find values for the following kinds of targets:

- Parameters of the controller action method that a request is routed to.
- Public properties of a controller, if specified by attributes.

http://localhost:8080/?NumberFromQueryString=2



```
0 references
public class HelloController : Controller
{
    [BindProperty]
    0 references | 0 exceptions
    public int NumberFromQueryString { get; set; }

    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
}
```



```
0 references
public class HelloController : Controller
{
    [BindProperty(SupportsGet = true)]
    0 references | 0 exceptions
    public int NumberFromQueryString { get; set; }

    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
}
```

Sources

By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:

1. Form fields
2. The request body (For controllers that have the [ApiController] attribute.)
3. Route data
4. Query string parameters
5. Uploaded files

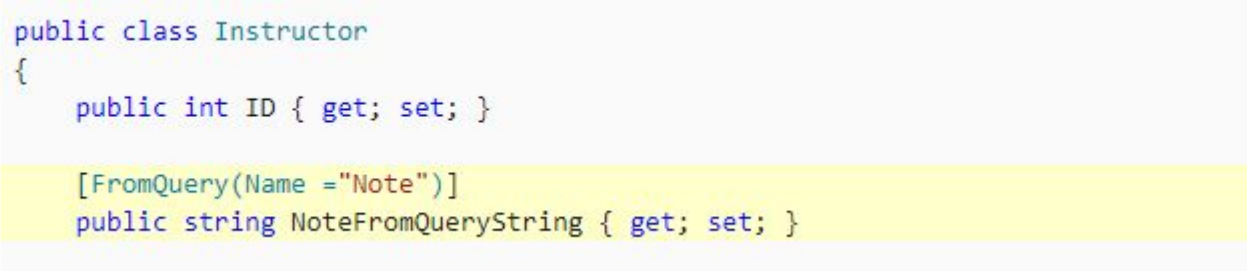
For each target parameter or property, the sources are scanned in the order indicated in this list. There are a few exceptions:

- Route data and query string values are used only for simple types.
- Uploaded files are bound only to target types that implement `IFormFile` or `IEnumerable<IFormFile>`.

Sources

If the default behavior doesn't give the right results, you can use one of the following attributes to specify the source to use for any given target.

- [FromQuery] - Gets values from the query string.
- [FromRoute] - Gets values from route data.
- [FromForm] - Gets values from posted form fields.
- [FromBody] - Gets values from the request body.
- [FromHeader] - Gets values from HTTP headers.



```
public class Instructor
{
    public int ID { get; set; }

    [FromQuery(Name ="Note")]
    public string NoteFromQueryString { get; set; }
```

No source for a model property

By default, a model state error isn't created if no value is found for a model property. The property is set to null or a default value:

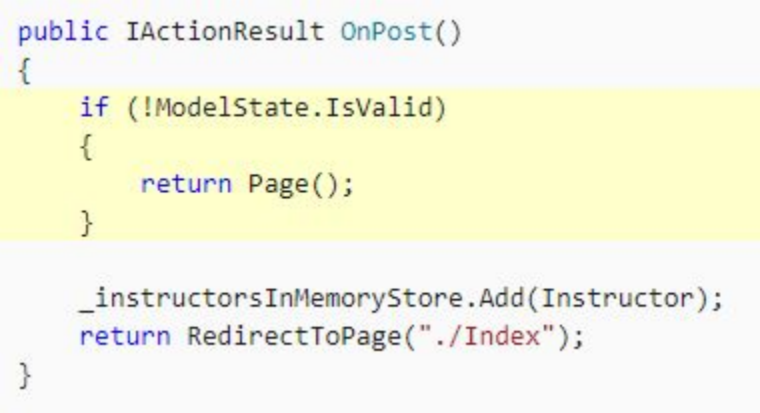
- Nullable simple types are set to `null`.
- Non-nullable value types are set to `default(T)`. For example, a parameter `int id` is set to 0.
- For complex Types, model binding creates an instance by using the default constructor, without setting properties.
- Arrays are set to `Array.Empty<T>()`, except that `byte[]` arrays are set to `null`.

Type conversions errors

If a source is found but can't be converted into the target type, model state is flagged as invalid. The target parameter or property is set to null or a default value, as noted in the previous section.

In an API controller that has the `[ApiController]` attribute, invalid model state results in an automatic HTTP 400 response.

In a Razor page, redisplay the page with an error message:



```
public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _instructorsInMemoryStore.Add(Instructor);
    return RedirectToPage("./Index");
}
```

Simple types

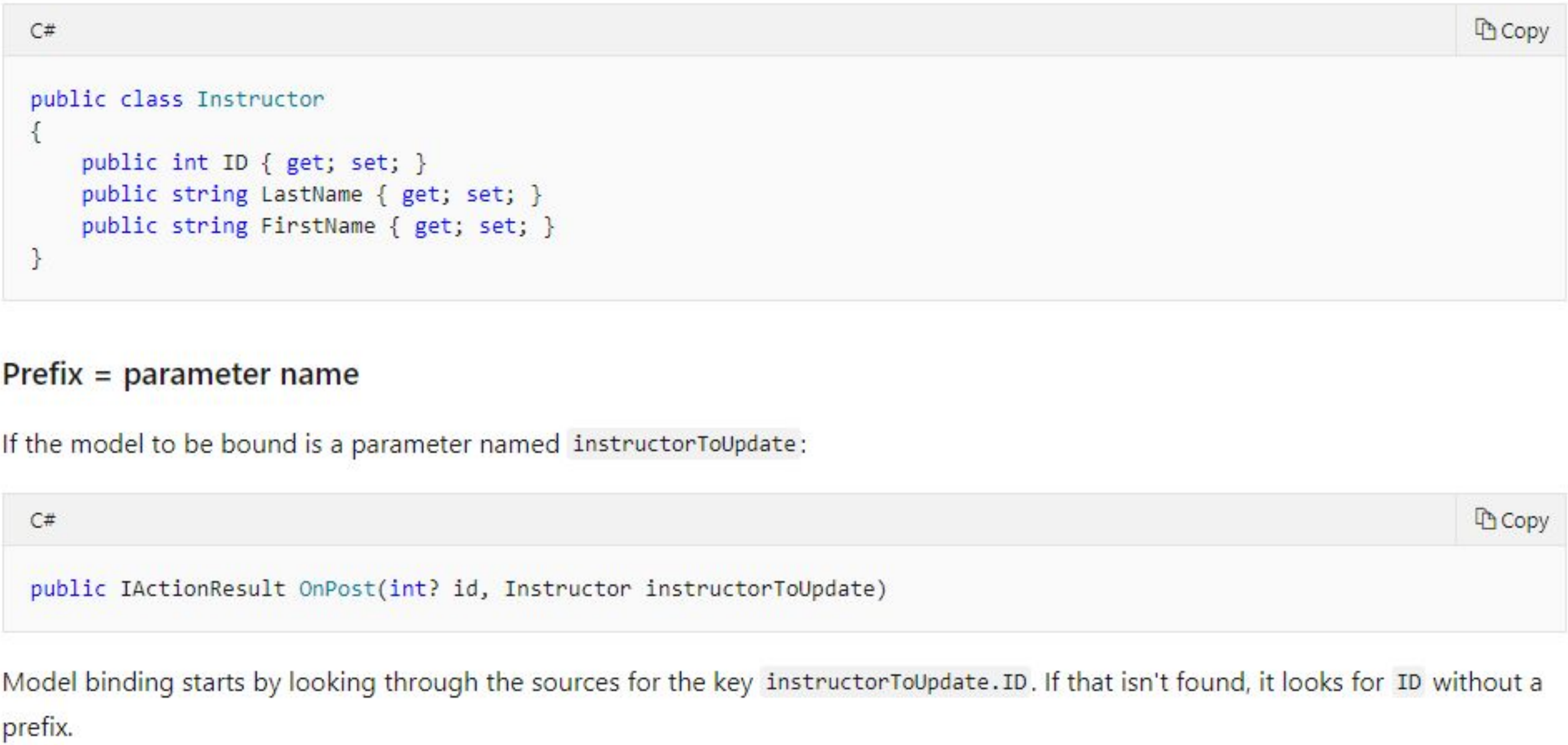
- Boolean
- Byte, SByte
- Char
- DateTime
- DateTimeOffset
- Decimal
- Double
- Enum
- Guid
- Int16, Int32, Int64
- Single
- TimeSpan
- UInt16, UInt32, UInt64
- Uri
- Version

Complex types

A complex type must have a public default constructor and public writable properties to bind. When model binding occurs, the class is instantiated using the public default constructor.

For each property of the complex type, model binding looks through the sources for the name pattern *prefix.property_name*. If nothing is found, it looks for just *property_name* without the prefix.

For binding to a parameter, the prefix is the parameter name. For binding to a `PageModel` public property, the prefix is the public property name. Some attributes have a `Prefix` property that lets you override the default usage of parameter or property name.



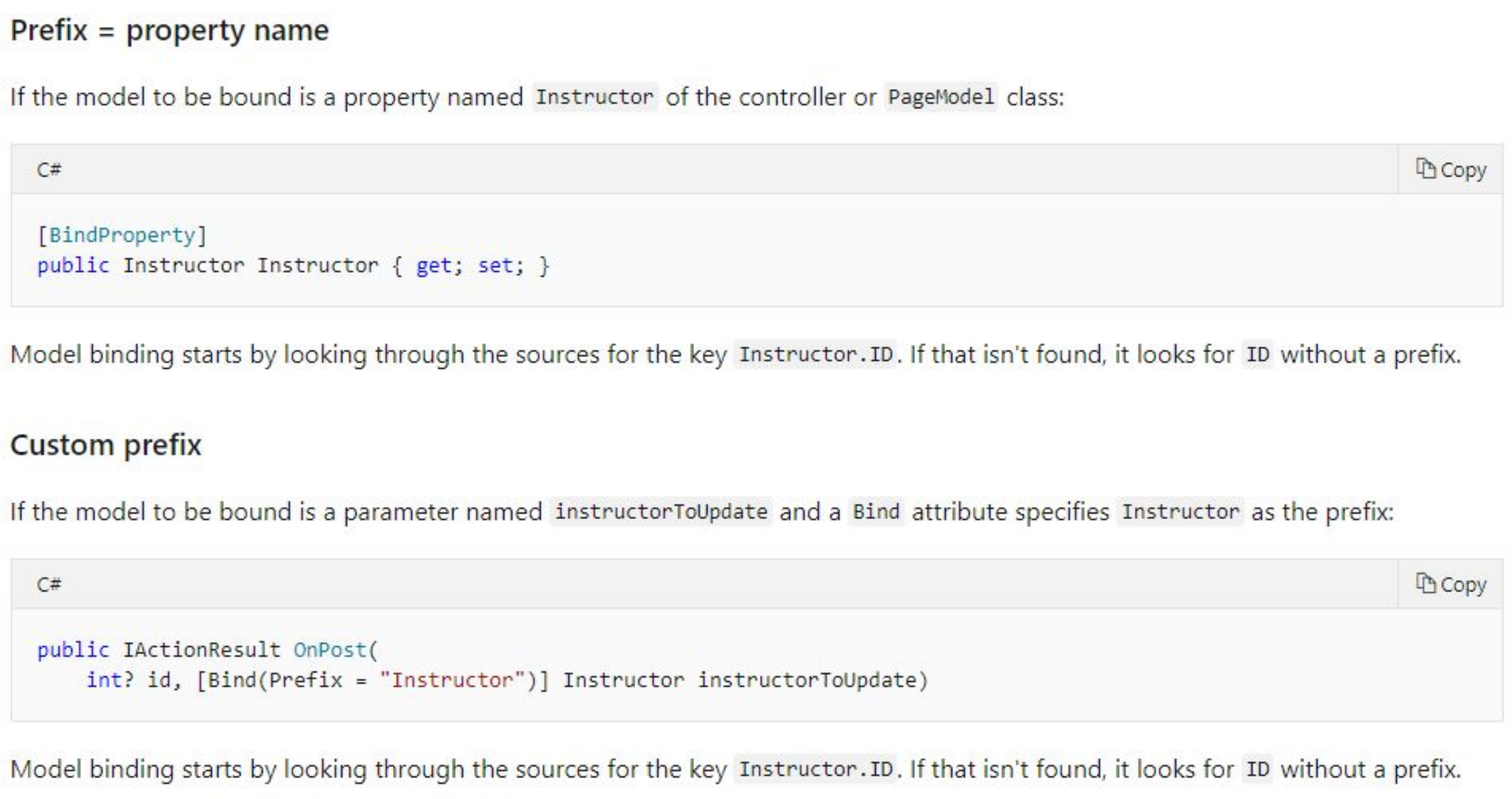
```csharp
public class Instructor
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Prefix = parameter name

If the model to be bound is a parameter named `instructorToUpdate`:

```csharp
public IActionResult OnPost(int? id, Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `instructorToUpdate.ID`. If that isn't found, it looks for `ID` without a prefix.



Prefix = property name

If the model to be bound is a property named `Instructor` of the controller or `PageModel` class:

```
[BindProperty]
public Instructor Instructor { get; set; }
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Custom prefix

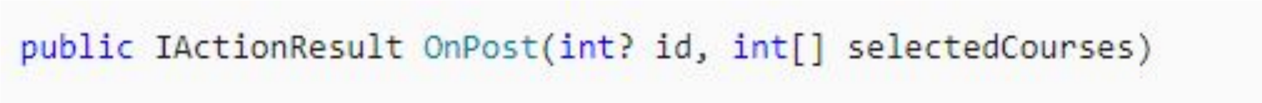
If the model to be bound is a parameter named `instructorToUpdate` and a `Bind` attribute specifies `Instructor` as the prefix:

```
public IActionResult OnPost(
    int? id, [Bind(Prefix = "Instructor")] Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Collections

For targets that are collections of simple types, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:



```
public IActionResult OnPost(int? id, int[] selectedCourses)
```

- Suppose the parameter to be bound is an array named `selectedCourses`:

```
selectedCourses=1050&selectedCourses=2000
```

```
selectedCourses[0]=1050&selectedCourses[1]=2000
```

```
[0]=1050&[1]=2000
```

```
[a]=1050&[b]=2000&index=a&index=b
```

```
selectedCourses[a]=1050&selectedCourses[b]=2000&selectedCourses.index=a&selectedCourses.index=b
```