Developing ASP.NET MVC Web Applications

Session: 3

Views in ASP.NET MVC

Objectives

- Define and describe views
- Explain and describe the razor engine
- Define and describe the HTML helper methods

Working with Views

- To display HTML content to the user, you can instruct the controller action in the application to return a view.
- A view:
  - Provides the UI of the application to the user.
  - Is used to display content of an application and also to accept user inputs.
  - Uses model data to create this UI.
  - Contains both HTML markup and code that runs on the Web server.

View Engines

- Are part of the MVC Framework that converts the code of a view into HTML markup that a browser can understand.
- Are divided in the following two categories:
    - **Web Form view engine**: Is a legacy view engine for views that use the same syntax as ASP.NET Web Forms.
    - **Razor view engine**: Is the default view engine starting from MVC 3. This view engine does not introduce a new programming language, but instead introduces new markup syntax to make transitions between HTML markups and programming code simpler.

Specifying the View for an Action 1-8

- While creating an ASP.NET MVC application, you often need to specify a view that should render the output for a specific action.
- When you create a new project in Visual Studio .NET, the project by default contains a Views directory.
- In an application, if a controller action returns a view, your application should contain the following:
  - A folder for the controller, with the same name as the controller without the `Controller` suffix.
  - A view file in the Home folder with the same name as the action.

Specifying the View for an Action 2-8

- Following code shows an `Index` action that returns an `ActionResult` object through a call to the `View()` method of the `Controller` class:

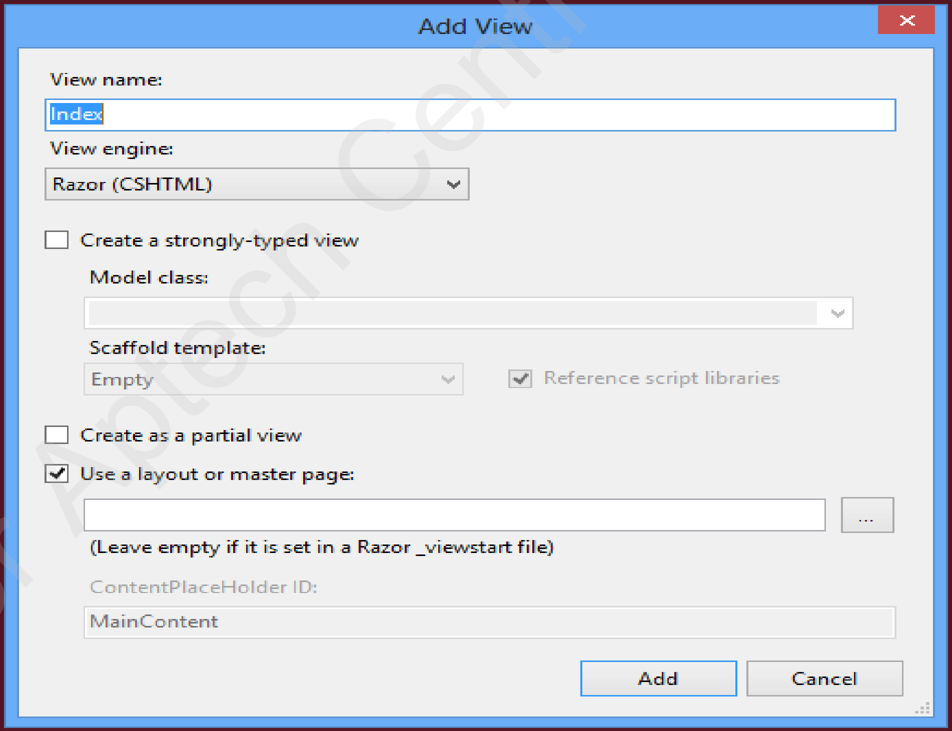
Code Snippet:

```
public class HomeController : Controller {
  public ActionResult Index()
  {
    return View();
  }
}
```

- In this code, the Index action of the controller named `HomeController` that returns the result of a call to the `View()` method. The result of the `View()` method is an `ActionResult` object that renders a view.

Specifying the View for an Action 3-8

- Visual Studio 2013 simplifies the process of creating a view.
- You can create the view file by performing the following steps:
  1. Right-click inside the action method for which you need to create a view.
  2. Select **Add View** from the context menu that appears. The **Add View** dialog box is displayed, as shown in the following figure:



Add View

View name: Index

View engine: Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template: Empty

☑ Reference script libraries

☐ Create as a partial view

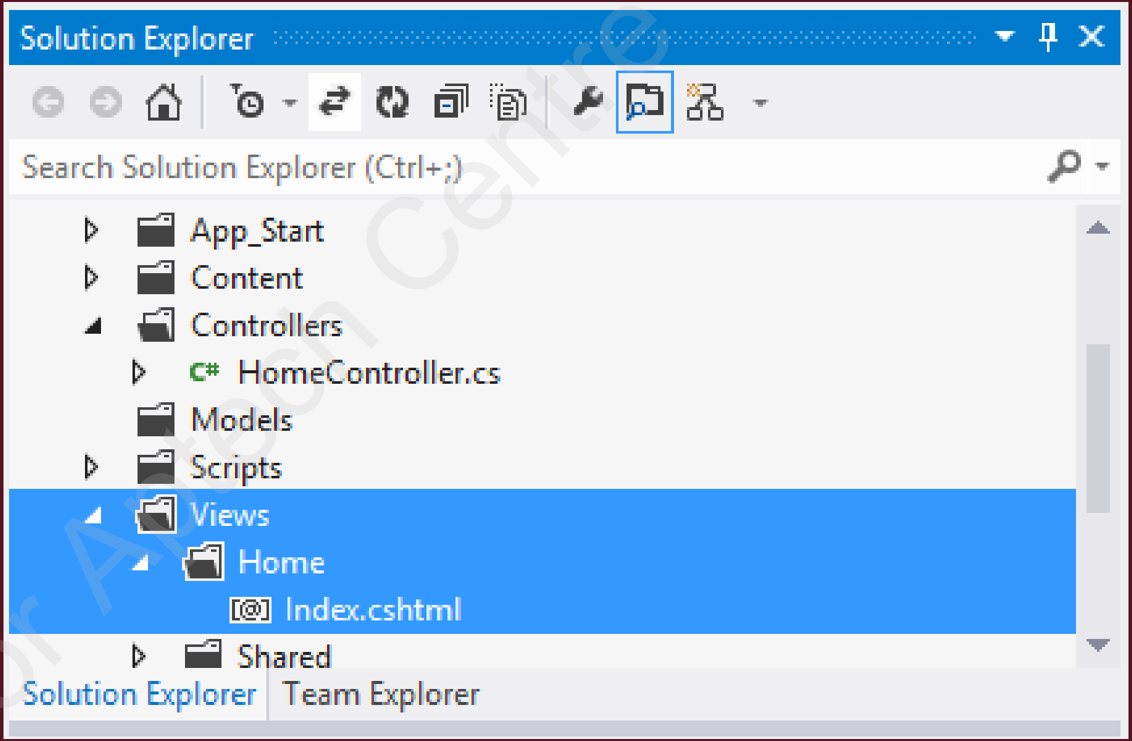
☑ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID: MainContent

Specifying the View for an Action 4-8

3. Click **Add**. Visual Studio.NET automatically creates an appropriate directory structure and adds the view file to it.
   - Following figure shows the view file that Visual Studio.NET creates for the `Index` action method of the `HomeController` class in the **Solution Explorer** window:



Specifying the View for an Action 5-8

- In the `Index.cshtml` file, you can add the following code that the view should display:

Code Snippet:

```
<!DOCTYPE html>
<html>
  <head>
        <title>Test View</title>
  </head>
  <body>
     <h1> Welcome to the Website  </h1>
  </body>
</html>
```

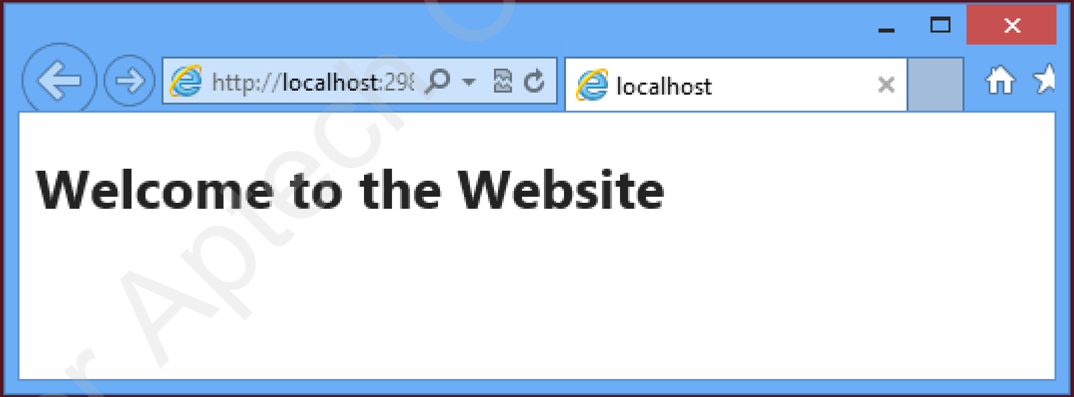
- This code creates a view with a title and a message as a heading.

Specifying the View for an Action 6-8

- When you access the `Index` action of the `HomeController` from a browser, the `Index.cshtml` view will be displayed.
- You can use the following URL to access the `Index` action of the `HomeController`:

```
http//localhost:1267/Home/Index
```

- Following figure shows the `Index.cshtml` view rendered in the browser:



Welcome to the Website

Specifying the View for an Action 7-8

- You can also render a different view from an action method.
- To return a different view, you need to pass the name of the view as a parameter.
- Following code shows how to return a different view:

Code Snippet:

```
public class HomeController : Controller
     {
       public ActionResult Index()
          {              return View("TestIndex");
          }
     }
```

- This code will search for a view inside the **/Views/Home** folder, but render the `TestIndex` view instead of rendering the `Index` view.

Specifying the View for an Action 8-8

- While developing an ASP.NET MVC application, you might also need to render a view that is present in a different folder instead of the default folder.
- To render such view, you need to specify the path to the view.
- Following code shows displaying a view named `Index.cshtml` present in the **/Views/Demo** folder:

Code Snippet:

```
public class HomeController : Controller
     {
       public ActionResult Index()
          {
              return View("~/Views/Demo/Welcome.cshtml");
          }
     }
```

- This code will display the view, named `Welcome.cshtml` defined inside the **/Views/Demo** folder.

Passing Data from a Controller to a View 1-14

- In an ASP.NET MVC application, a controller typically performs the business logic of the application and needs to return the result to the user through a view.
- You can use the following objects to pass data between controller and view:
  - `ViewData`
  - `ViewBag`
  - `TempData`

Passing Data from a Controller to a View 2-14

- **ViewData**
  - Passes data from a controller to a view.
  - Is a dictionary of objects that is derived from the `ViewDataDictionary` class.
  - Some of the characteristics of `ViewData` are as follows:
    - The life of a `ViewData` object exists only during the current request.
    - The value of `ViewData` becomes null if the request is redirected.
    - `ViewData` requires typecasting when you use complex data type to avoid error.
  - The general syntax of `ViewData` is as follows:

Syntax:

```
ViewData[<key>] = <Value>;
```

where,

- `Key`: Is a `String` value to identify the object present in `ViewData`.
- `Value`: Is the object present in `ViewData`. This object may be a `String` or a different type, such as `DateTime` .

Passing Data from a Controller to a View 3-14

- Following code shows a `ViewData` with two key-value pairs in the `Index` action method of the `HomeController` class:

Code Snippet:

```
public class HomeController : Controller {
public ActionResult Index()
 {
  ViewData["Message"] = "Message from ViewData";
   ViewData["CurrentTime"] = DateTime.Now;return View();
   }
}
```

- In this code a `ViewData` is created with two key-value pairs.
- The first key named `Message` contains the `String` value, `Message from ViewData`.
- The second key named `CurrentTime` contains the value, `DateTime.Now`.

Passing Data from a Controller to a View 4-14

- Following code shows retrieving the values present in `ViewData`:

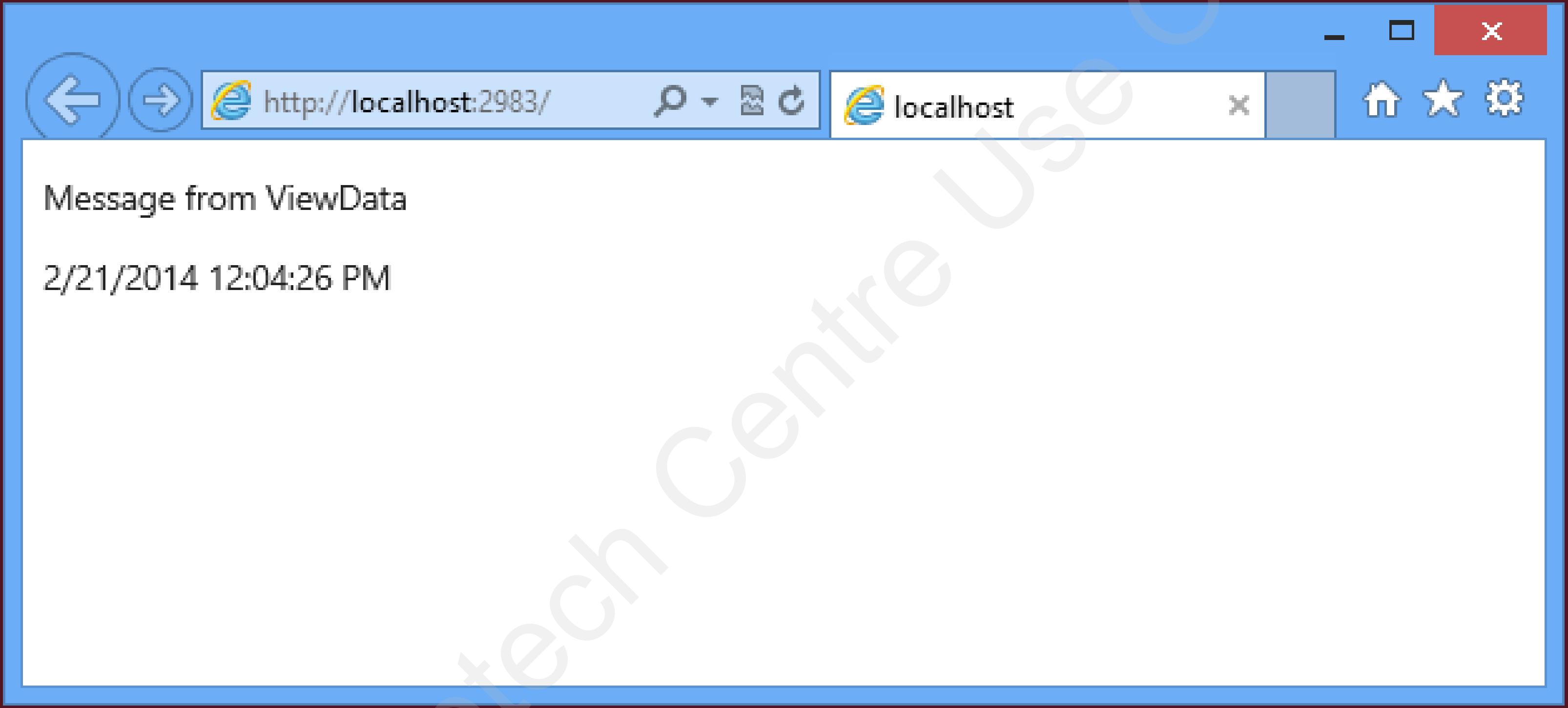
Code Snippet:

```
<!DOCTYPE html>
<html>
  <head>
        <title>Index View</title>
  </head>
  <body>
    <p> @ViewData["Message"] </p>
    <p> @ViewData["CurrentTime"] </p>
  </body>
</html>
```

- In this code, `ViewData` is used to display the values of the `Message` and `CurrentTime` keys.

Passing Data from a Controller to a View 5-14

- Following figure shows the shows the output of the `ViewData`:



```
Message from ViewData

2/21/2014 12:04:26 PM
```

Passing Data from a Controller to a View 6-14

- **`ViewBag`**:
  - Is a wrapper around `ViewData`.
  - Exists only for the current request and becomes `null` if the request is redirected.
  - Is a dynamic property based on the dynamic features introduced in C# 4.0.
  - Does not require typecasting when you use complex data type.
- The general syntax of `ViewBag` is as follows:

Syntax:

```
ViewBag.<Property> = <Value>;
```

where,

- `Property`: Is a `String` value that represents a property of `ViewBag`.
- `Value`: Is the value of the property of `ViewBag`.

Passing Data from a Controller to a View 7-14

- Following code shows a ViewBag with two properties in the Index action method of the HomeController class:

Code Snippet:

```
public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Message from ViewBag";
            ViewBag.CurrentTime = DateTime.Now;
            return View();
        }
    }
```

- In this code a ViewBag is created with the following two properties:
    - The first property named Message contains the String value, 'Message from ViewBag'.
    - The second property named CurrentTime contains the value, DateTime.Now.

Passing Data from a Controller to a View 8-14

- Following code shows retrieving the values present in `ViewBag`:

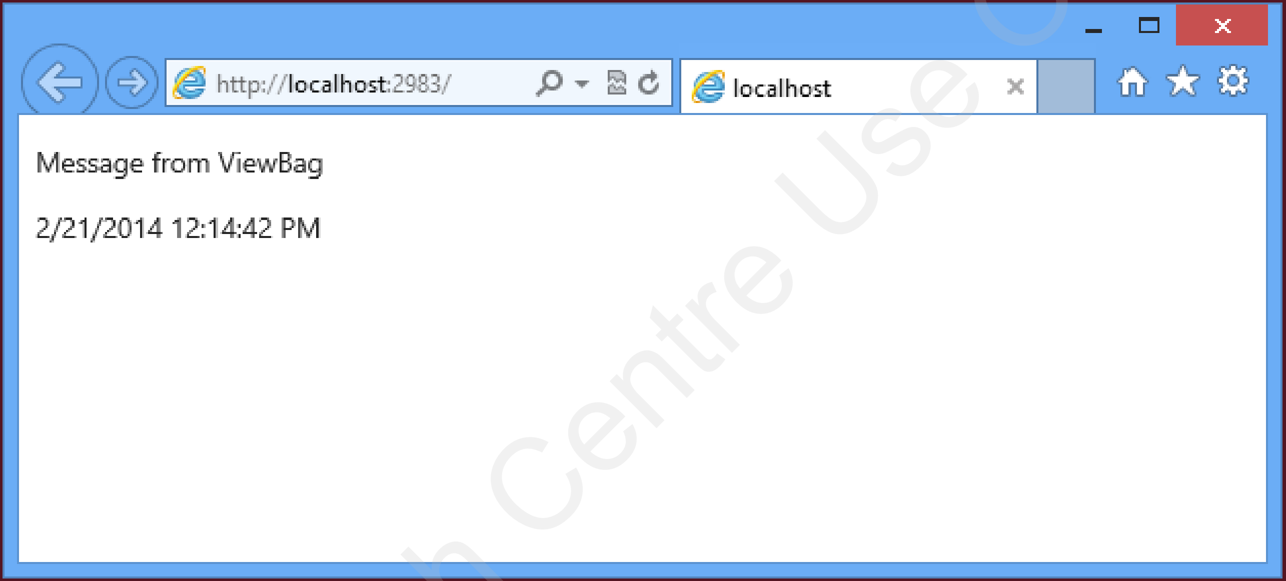
Code Snippet:

```
<!DOCTYPE html>
<html>
  <head>
        <title>Index View</title>
  </head>
  <body>
    <p>
        @ViewBag.Message
    </p>
    <p>
         @ViewBag.CurrentTime
     </p>
  </body>
</html>
```

- In this code, `ViewBag` is used to display the values of `Message` and `CurrentTime` properties.

Passing Data from a Controller to a View 9-14

- Following figure shows the output of the `ViewBag`:



```
Message from ViewBag

2/21/2014 12:14:42 PM
```

- When you use a `ViewBag` to store a property and its value in an action, that property can be accessed by both `ViewBag` and `ViewData` in a view.

Passing Data from a Controller to a View 10-14

- Following code shows a controller action storing a `ViewBag` property:

Code Snippet:

```
public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.CommonMessage = "Common message accessible to both
ViewBag and ViewData";
             return View();
        }
    }
```

- In this code, a `ViewBag` is created with a property named `CommonMessage`.

Passing Data from a Controller to a View 11-14

- Following code shows a view that uses both `ViewData` and `ViewBag` to access the `CommonMessage` property stored in `ViewBag`:

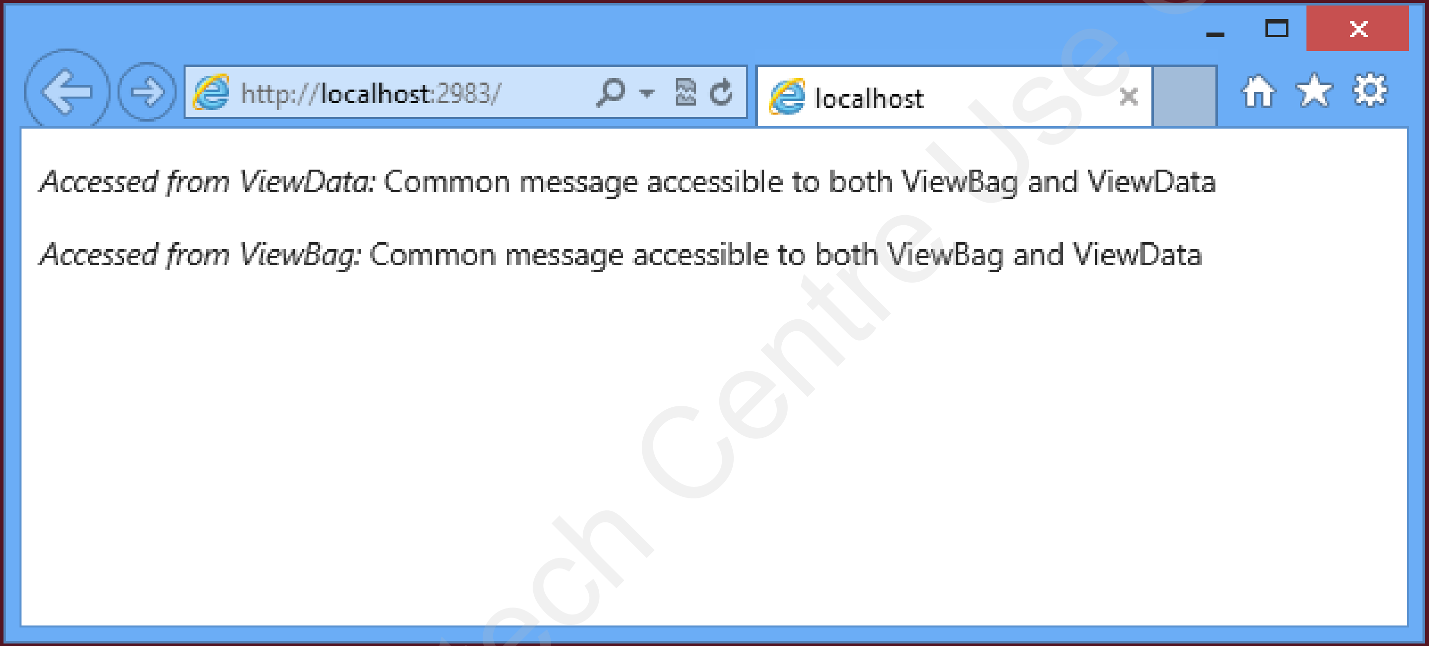
Code Snippet:

```
<!DOCTYPE html>
<html>
  <head>
        <title>Index View</title>
  </head>
  <body>
    <p>
      <em>Accessed from ViewData:</em> @ViewData["CommonMessage"]
    </p>
    <p>
       <em>Accessed from ViewBag:</em> @ViewBag.CommonMessage
     </p>
  </body>
</html>
```

- This code uses both `ViewData` and `ViewBag` to display the value of the `CommonMessage` property stored in `ViewBag`.

Passing Data from a Controller to a View 12-14

- Following figure shows output of `ViewData` and `ViewBag`:



Accessed from ViewData: Common message accessible to both ViewBag and ViewData

Accessed from ViewBag: Common message accessible to both ViewBag and ViewData

Passing Data from a Controller to a View 13-14

- **TempData**:
    - Is a Dictionary object derived from the TempDataDictionary class.
    - Stores data as key-value pairs.
    - Allows passing data from the current request to the subsequent request during request redirection.

The general syntax of TempData is as follows:

Syntax:

```
TempData[<Key>] = <Value>;
```

where,

- Key: Is a String value to identify the object present in TempData.
- Value: Is the object present in TempData.

Passing Data from a Controller to a View 14-14

- Following code shows how to use `TempData` to pass values from one view to another view through request redirection:

Code Snippet:

```
public class HomeController : Controller {
        public ActionResult Index() {
            ViewData["Message"] = "ViewData Message";
            ViewBag.Message = "ViewBag Message";
            TempData["Message"] = "TempData Message";
            return Redirect("Home/About");
        }
        public ActionResult About() {
            return View();
        }
    }
```

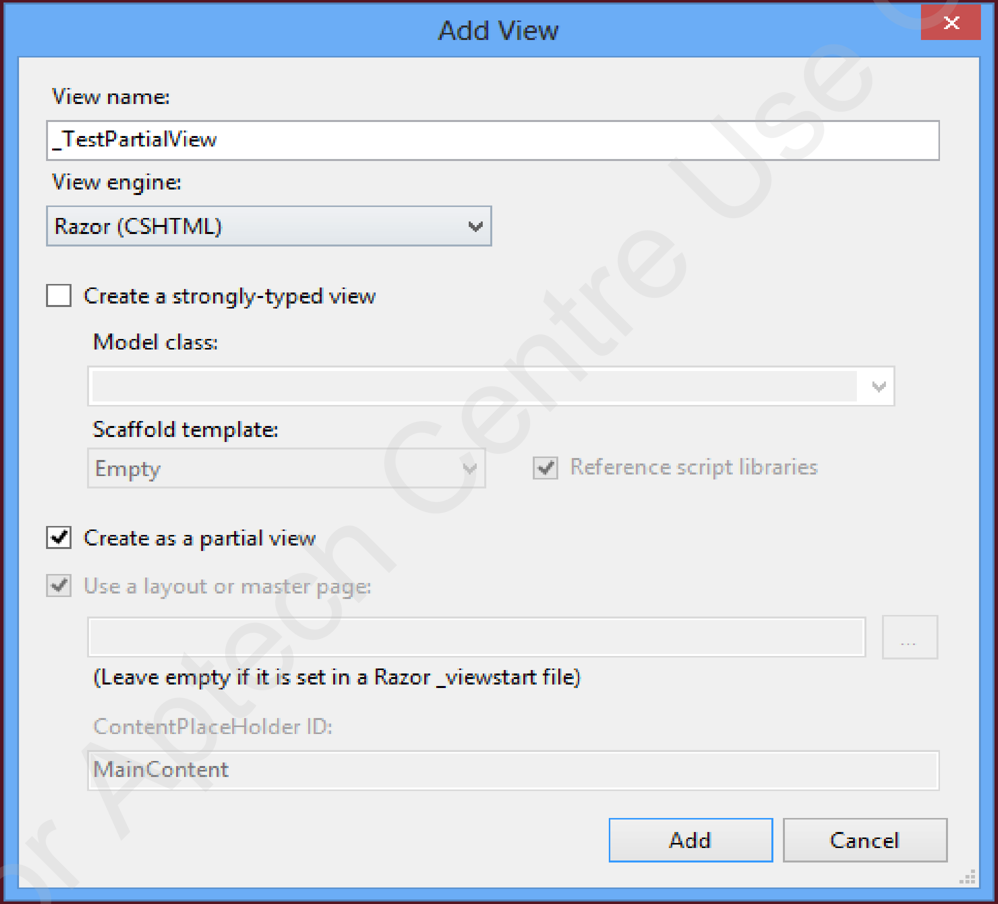
- This code creates two actions. The `Index` action stores value to `ViewData`, `ViewBag`, and `TempData` objects. Then, redirects the request to the About action by calling the `Redirect()` method. The `About` Action returns the corresponding view, which is the `About.cshtml` view.

Using Partial Views 1-5

- In an ASP.NET MVC application, a partial view:
  - Represents a sub-view of a main view.
  - Allows you to reuse common markups across the different views of the application.
- To create a partial view in Visual Studio .NET, you need to perform the following steps:
  1. Right-click the **Views/Shared** folder in the **Solution Explorer** window and select **Add→View**. The **Add** View dialog box is displayed.
  2. In the **Add View** dialog box, specify a name for the partial view in the **View Name** text field.
  3. Select the **Create as a partial view** check box.

Using Partial Views 2-5

- Following figure shows how to create a partial view in Visual Studio 2013:



Add View

View name: _TestPartialView

View engine: Razor (CSHTML)

- [ ] Create a strongly-typed view

Model class:

Scaffold template: Empty

- [x] Reference script libraries
- [x] Create as a partial view
- [x] Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID: MainContent

4. Click **Add** button to create the partial view.

Using Partial Views 3-5

- In the partial view, you can add the markup that you need to display in the main view, as shown in the following code:

```
<h3> Content of partial view. </h3>
```

- The general syntax of partial view is as follows:

Syntax:

```
@Html.Partial(<partial_view_name>)
```

where,

- `partial_view_name`: Is the name of the partial view without the `.cshtml` file extension.

Using Partial Views 4-5

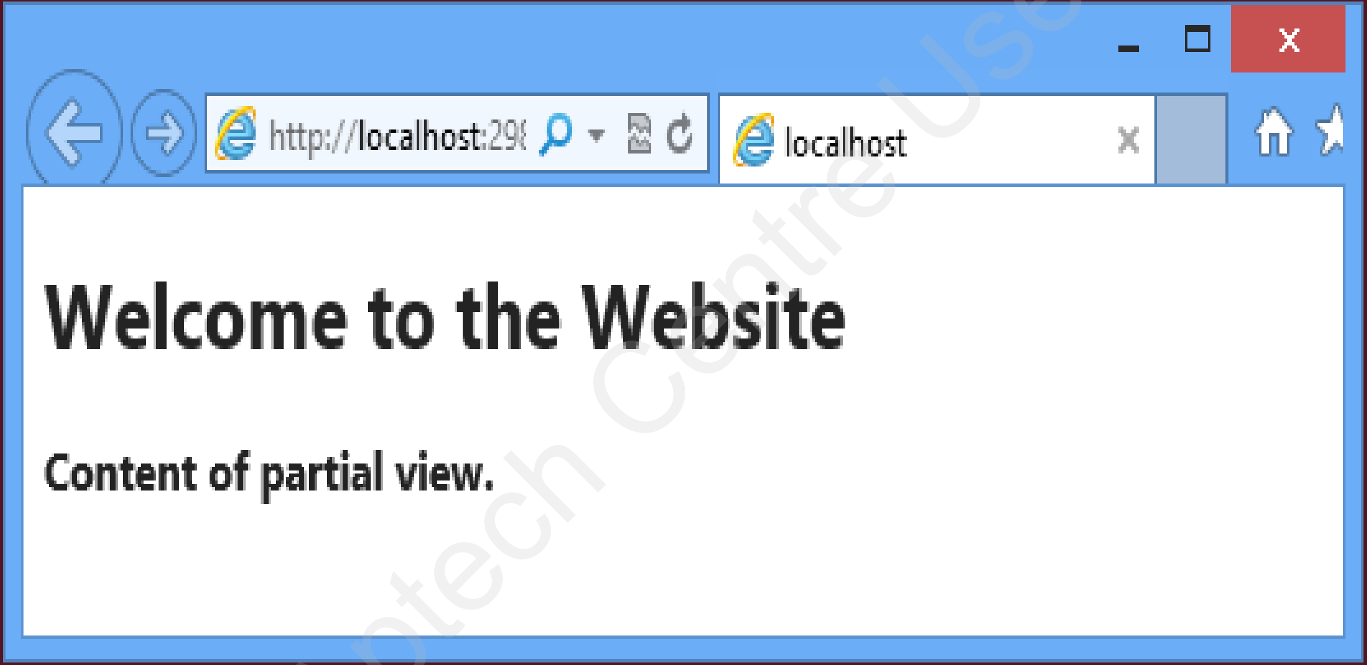
- Following code shows a main view, named `Index.cshtml` that accesses the partial view, named `_TestPartialView.cshtml`:

Code Snippet:

```
<!DOCTYPE html>
<html>
  <head>
        <title>Index View</title>
  </head>
  <body>
    <h1>
      Welcome to the Website
    </h1>
    <div>@Html.Partial("_TestPartialView")</div>
  </body>
</html>
```

- This code shows the markup of the main `Index.cshtml` view that displays a welcome message and renders the partial view, named `_TestPartialView.cshtml`.

- Following figure shows the output of the main view, named `Index.cshtml`:



Welcome to the Website

Content of partial view.

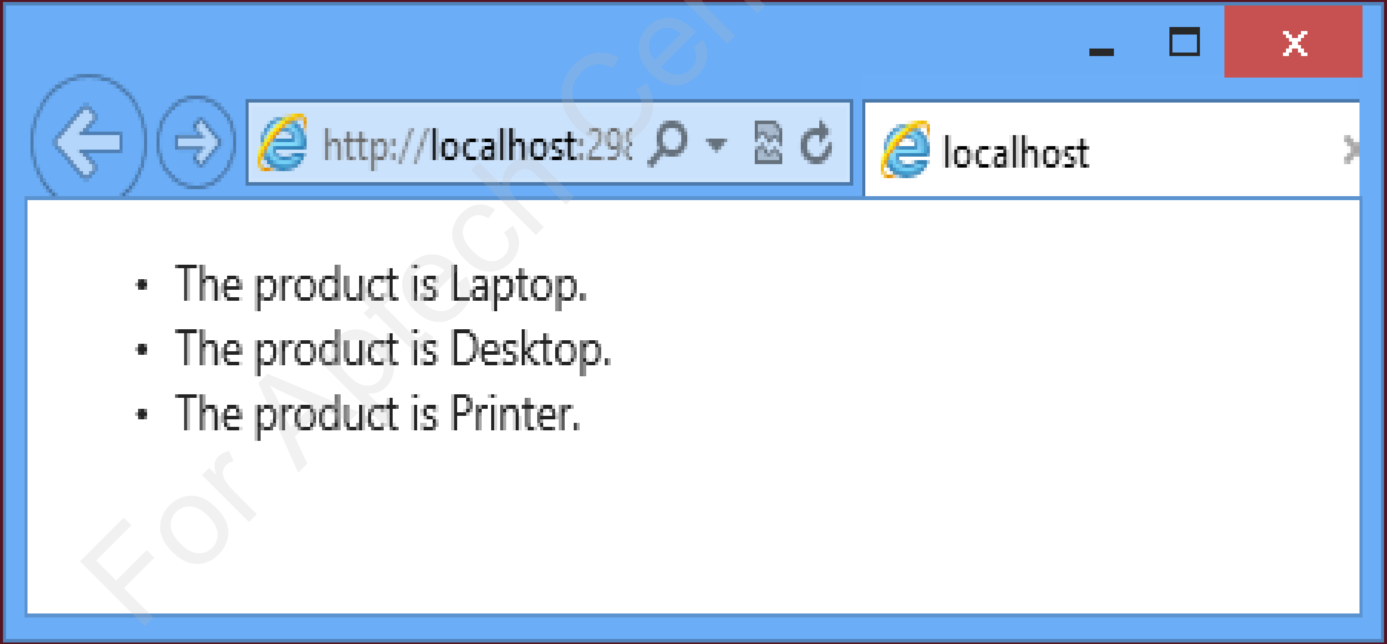
Razor 1-2

- **Razor**:
    - Is a syntax, based on the ASP.NET Framework that allows creating views.
    - Is simple and easy to understand for users who are familiar with the C#.NET or VB.NET programming languages.
    - Following code snippet shows a simple razor view containing both HTML markups for presentation and C# code:

Code Snippet:

```
@{var products = new string[] {"Laptop", "Desktop", "Printer"};}
<html>
<head><title>Test View</title></head>
<body>
<ul>
@foreach (var product in products){
<li>The product is @product.</li>}
</ul>
</body>
</html>
```

- In the preceding code:
  - A `string[]` is declared and initialized using Razor syntax.
  - Then, Razor syntax is used to iterate through the elements of the array and display each element.
  - The remaining code in the view is HTML code that displays the page title, body, and the array elements in a bullet list.
- Following figure shows the output of simple razor view:



```
• The product is Laptop.
• The product is Desktop.
• The product is Printer.
```

Razor Engine

- The MVC Framework uses a view engine to convert the code of a view into HTML markup that a browser can understand.
- Razor engine:
  - Is used as the default view engine by the MVC Framework.
  - Compiles a view of your application when the view is requested for the first time.
  - Delivers the compiled view for subsequent requests until you make changes to the view.
  - Does not introduce a new set of programming language, but provides template markup syntax to segregate HTML markup and programming code in a view.
  - Supports Test Driven Development (TDD) which allows you to independently test the views of an application.

Razor Syntax Rules 1-5

- A Razor:
  - First requires identifying the server-side code from the markup code to interpret the server-side code embedded inside a view file.
  - Uses the `@` symbol, to separate the server-side code from the markup code.
- While creating a Razor view, you should consider following rules:
  - Enclose code blocks between `@{` and `}`
  - Start inline expressions with `@`
  - Variables are declared with the `var` keyword
  - Enclose strings in quotation marks
  - End a Razor code statement with semicolon (;)
  - Use the `.cshtml` extension to store a Razor view file that uses C# as the programing language
  - Use the `.vbhtml` extension to store a Razor view file that uses VB as the programing language.

Razor Syntax Rules 2-5

- Razor supports code blocks within a view. A code block is a part of a view that only contains code written in C# or VB.
- The general syntax of using Razor is as follows:

Syntax:

```
@{   <code>}
```

where,

- code: is the C# or VB code that will execute on the server.

- Following code snippet shows two single-statement code blocks in a Razor view:

Code Snippet:

```
@{ var myMessage = "Hello World"; }
@{ var num = 10;         }
```

- This code shows two single-statement that declares the variables, `myMessage` and `num`. The `@{` characters mark the beginning of each code and the `}` character marks the end of the code.

Razor Syntax Rules 3-5

- Razor also supports multi-statement code blocks where each code block can have multiple statements.
- Multi-statement code block allows you to ignore the use of the `@` symbol in every line of code.
- Following code snippet shows a multi-statement code block in a Razor view:

Code Snippet:

```
@{
var myMessage = "Hello World";
var num = 10;
}
```

- This code shows a multi-statement code block that declares the variables, `myMessage` and `num`.

Razor Syntax Rules 4-5

- Similar to code block, Razor uses the `@` character for an inline expression.
- Following code snippet shows using inline expressions:

Code Snippet:

```
@{
var myMessage = "Hello World";
var num = 10;
}
@myMessage is numbered @num.
```

  - This code uses two inline expressions that evaluates the `myMessage` and `num` variables.
  - When the Razor engine encounters the `@` character, it interpreted the immediately following variable name as server-side code and ignores the following text.

Razor Syntax Rules 5-5

- At times, you may also require overriding the logic that Razor uses to identify the server-side code.
- Following code snippet shows including the @ symbol in an e-mail address:

Code Snippet:

```
<h3>The email ID is: john@@mvcexample.com <h3>
```

- In this code, Razor:
  - Interprets the first @ symbol as an escape sequence character.
  - Uses an implicit code expression to identify the part of a server-side code.
- Sometimes, Razor may also interpret an expression as markup instead of running as server-side code.

Variables 1-2

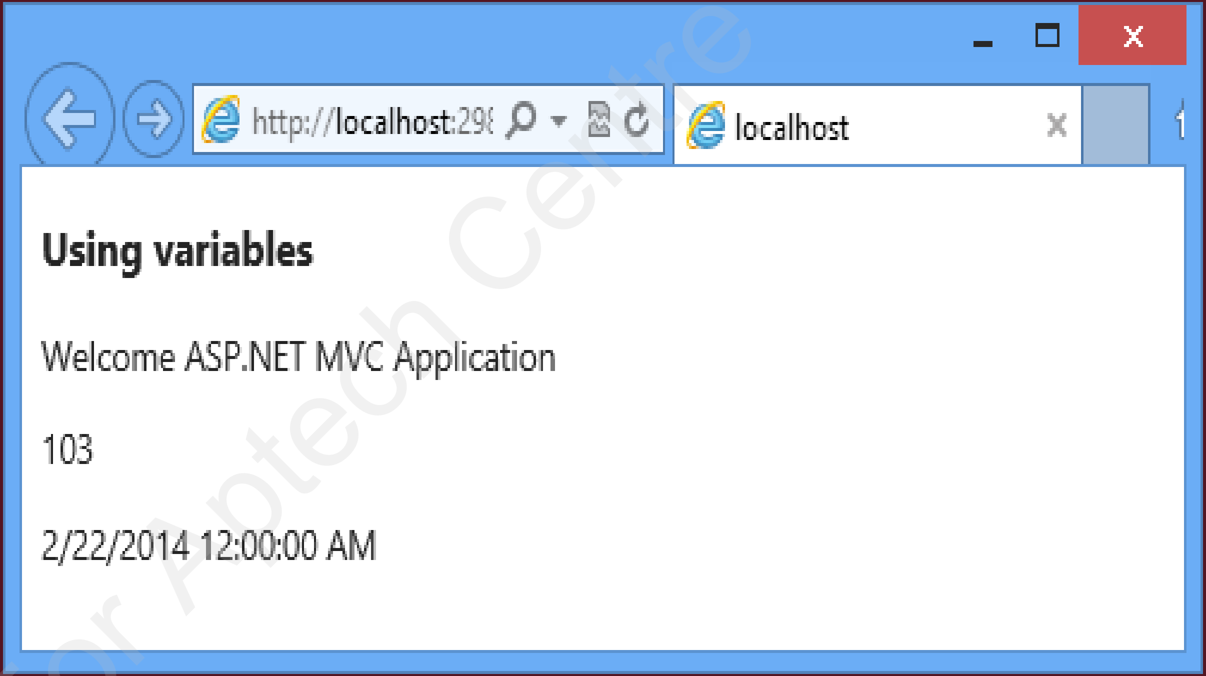
- Are used to store data.
- In Razor, you declare and use a variable in the same way as you do in a C# program.
- Following code snippet shows declaring variables using Razor:

Code Snippet:

```
<!DOCTYPE html>
<html><body>
@{
    var heading = "Using variables";
    string greeting = "Welcome ASP.NET MVC Application";
    int num = 103;
    DateTime today = DateTime.Today;
    <h3>@heading</h3>
    <p>@greeting</p>
    <p>@num</p>
    <p>@today</p>
    }
</body></html>
```

Variables 2-2

- The preceding code shows declaring four variables, such as `heading`, `greeting`, `num`, and `today` using Razor.
- Following figure shows the output of declaring variables using Razor:



```
Using variables

Welcome ASP.NET MVC Application

103

2/22/2014 12:00:00 AM
```

Loops 1-4

- While developing an ASP.NET MVC Web application, you might require executing same statement continually.
- In such scenario you can use loops.
- C# supports the following four types of loop constructs:
  - The `while` loop
  - The `for` loop
  - The `do…while` loop
  - The `foreach` loop

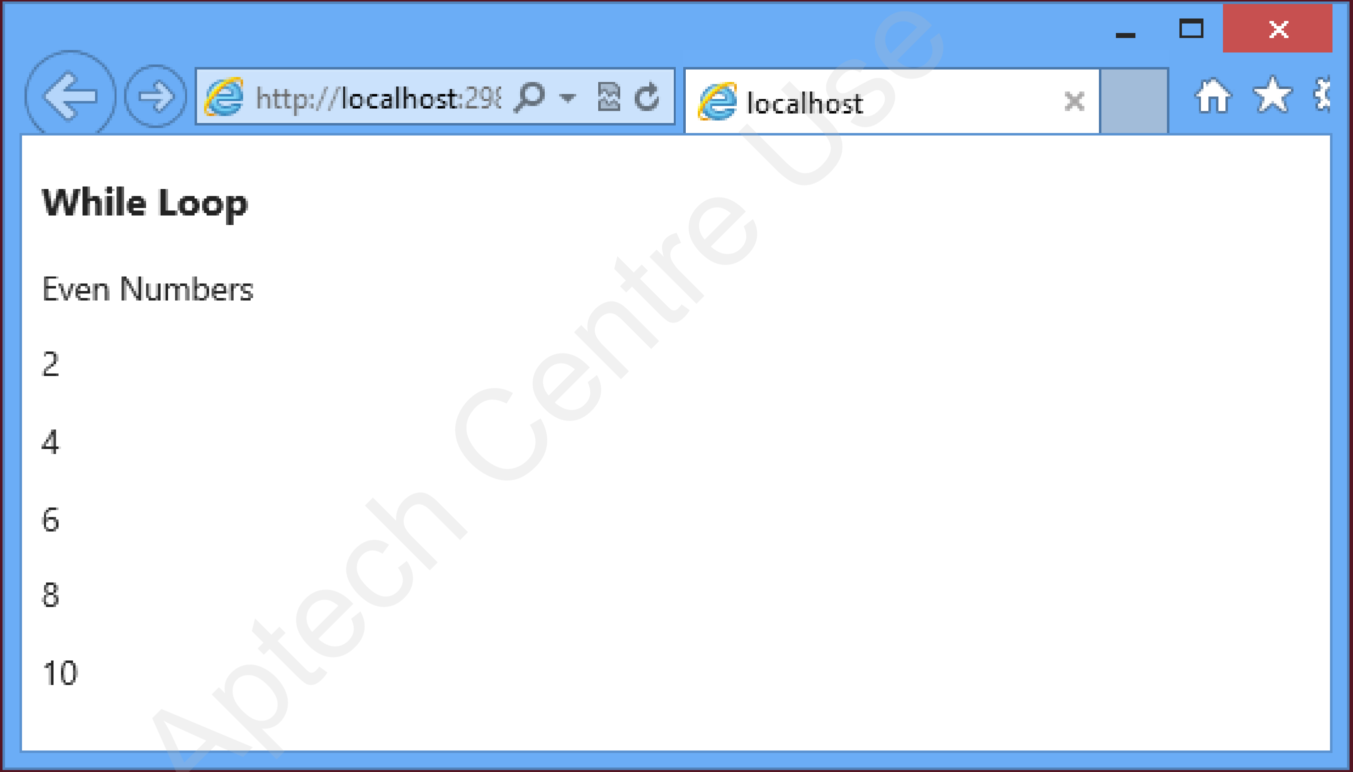
Loops 2-4

- `while` loop:
    - Is used to execute a block of code repetitively as long as the condition of the loop remains true.
    - Uses the `while` keyword at the beginning followed by parentheses.
    - Allows you to specify the number of time the loop continues inside the parentheses, then, a block to repeat.
    - Following code snippet shows the Razor code that uses a `while` loop to print the even numbers from 1 to 10:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
@{ var b = 0;
while (b < 7) {
    b += 1;
   <p>Text @b</p>   }
}
</body>
</html>
```

- Following figure shows the output of the razor code using `while` loop:



While Loop

Even Numbers

2

4

6

8

10

Loops 4-4

- When you know the number of time that a statement should execute, you can use a for loop.
- Similar to the `while` loop, you can use the `for` loop to iterate through elements.
- Following code snippet shows the Razor code that uses a `for` loop to print the even numbers from 1 to 10:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
    <h1>Even Numbers</h1>
@{ var num=1;
for (num = 1; num <= 11; num++){
    if ((num % 2) == 0)
     { <p> @num</p>  }
  } }
</body>
</html>
```

- The code shows the for loop to print the even numbers from 1 to 10 using Razor.

Conditional Statements 1-4

- Allows you to display dynamic content based on some specific conditions.
- Includes the `if` statement that returns `true` or `false`, based on the specified condition.
- Following code snippet shows using the `if...else` statements using Razor:

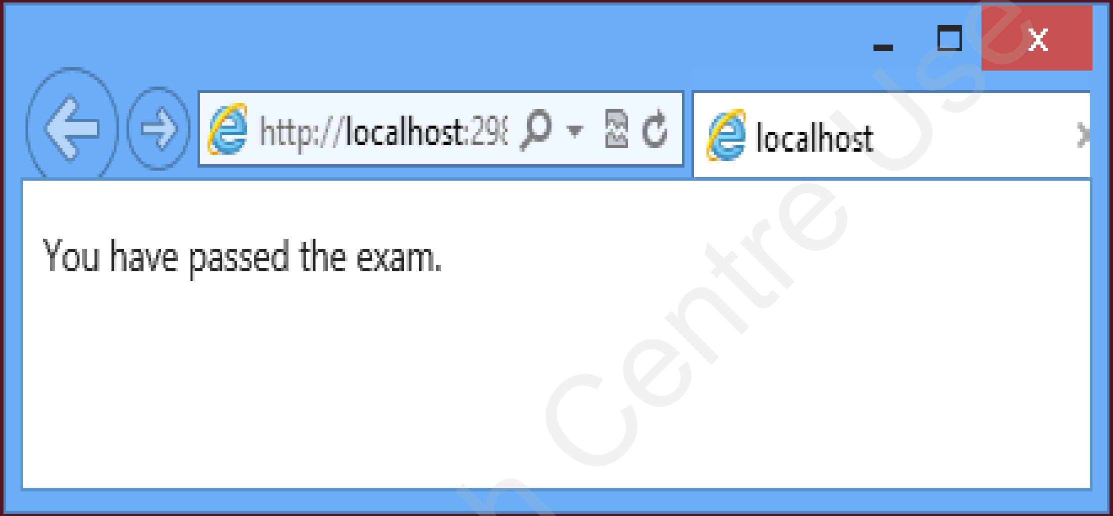
Code Snippet:

```
<!DOCTYPE html>
@{var mark=60;}
<html>
<body>
@if (mark>80)  {
  <p>You have failed in the exam.</p> }
else  {
  <p>You have passed the exam.</p>   }
</body>
</html>
```

- The code uses the `if...else` statements using Razor.

Conditional Statements 2-4

- Following figure shows the output of the code using the `if...else` statements using Razor:



You have passed the exam.

- You can also test a number of individual conditions in your code by using a `switch` statement in the view.

Conditional Statements 3-4

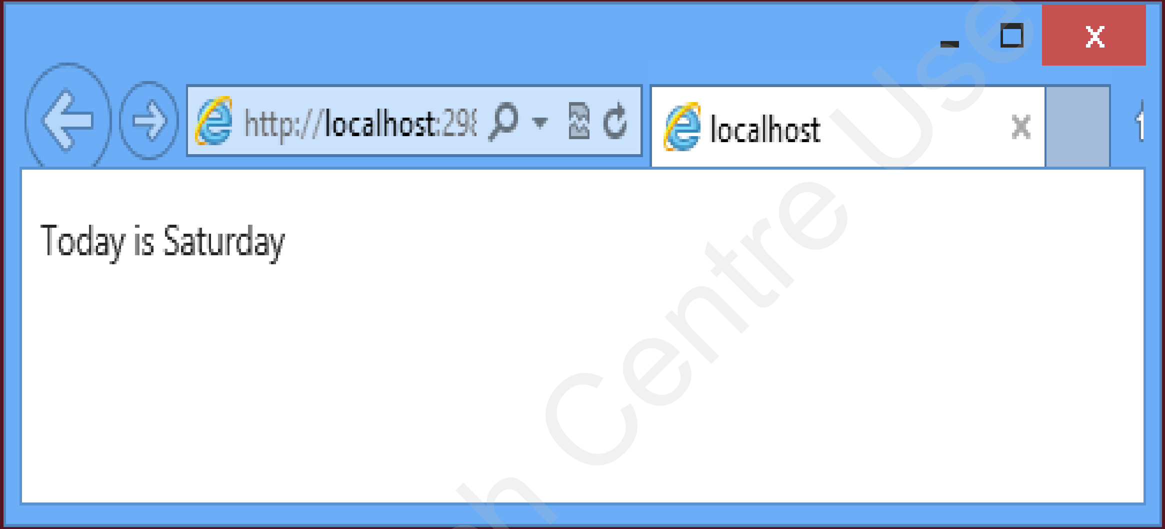
- Following code snippet shows using the `switch` statement using Razor:

Code Snippet:

```
<!DOCTYPE html>
@{
var day=DateTime.Now.DayOfWeek.ToString();
var msg="";   }
   <html> <body>
   @switch(day)     {
      case "Monday":
       msg="Today is Monday, the first working day.";
      break;
     case "Friday":
       msg="Today is Friday, the last working day.";
     break;
    default:
      msg="Today is " + day;
      break; }
<p>@msg</p> </body> </html>
```

Conditional Statements 4-4

- Following figure shows the output of the code using switch statement:



```
Today is Saturday
```

Arrays 1-2

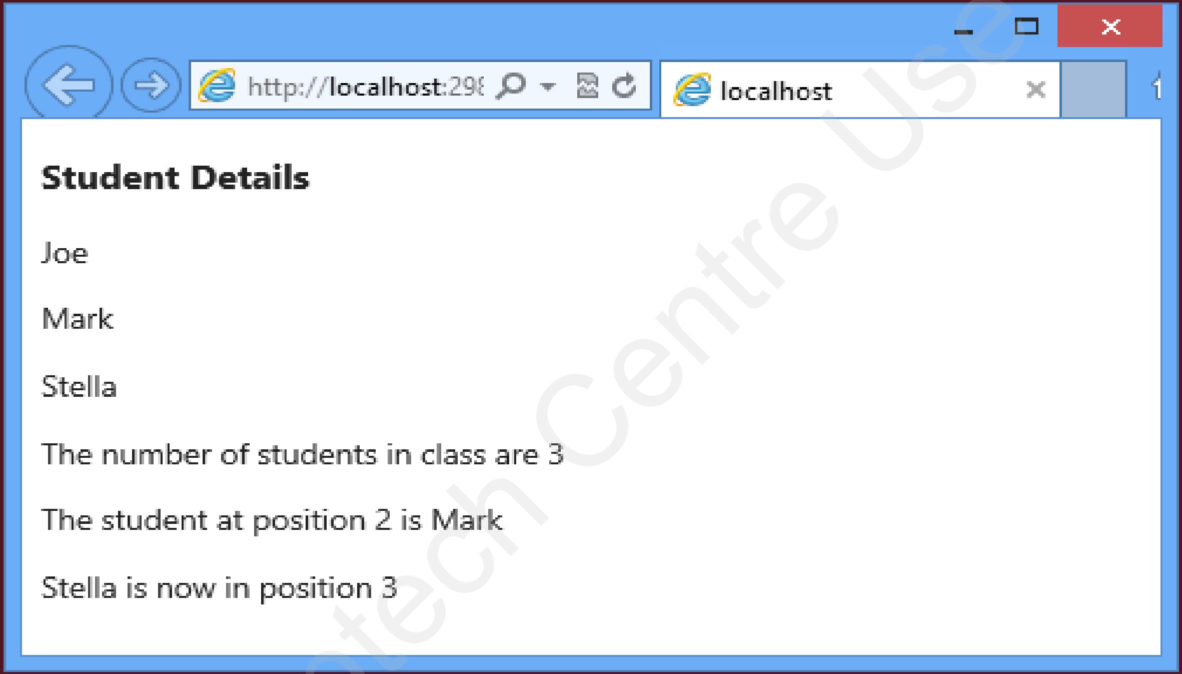
- Allows you to store similar variables.
- Following code snippet shows using arrays to store similar variables using Razor:

Code Snippet:

```
<!DOCTYPE html>
@{
string[] members = {"Joe", "Mark", "Stella"};
int i = Array.IndexOf(members, "Stella")+1;
int len = members.Length;
string x = members[2-1]; }
<html> <body>
<h3>Student Details</h3>
@foreach (var person in members) {
<p>@person</p> }
<p>The number of students in class are @len</p>
<p>The student at position 2 is @x</p>
<p>Stella is now in position @i</p>
</body>
</html>
```

Arrays 2-2

- Following figure shows the output of arrays that stores similar variables:



Student Details

Joe

Mark

Stella

The number of students in class are 3

The student at position 2 is Mark

Stella is now in position 3

HTML Helper Methods 1-17

- In ASP.NET MVC Framework, helper methods:
  - Are extension methods to the `HtmlHelper` class, can be called only from views.
  - Simplifies the process of creating a view.
  - Allows generating HTML markup that you can reuse across the Web application.
- Some of the commonly used helper methods while developing an MVC application are as follows:
  - `Html.ActionLink()`
  - `Html.BeginForm()` and `Html.EndForm()`
  - `Html.Label()`
  - `Html.TextBox()`
  - `Html.TextArea()`
  - `Html.Password()`
  - `Html.CheckBox()`

HTML Helper Methods 2-17

- `Html.ActionLink()` helper method allows you to generate a hyperlink that points to an action method of a controller class.
- The general syntax of the `Html.ActionLink()` helper method is as follows:

Syntax:

```
@Html.ActionLink(<link_text>,<action_method>,
<optional_controller>)
```

where,

- `link_text`: Is the text to be displayed as a hyperlink.
- `action_method`: Is the name of the action method that acts as the target for the hyperlink
- `optional_controller`: Is the name of the controller that contains the action method that will get called by the hyperlink. This parameter can be omitted if the action method getting called is in the same controller as the action method whose view renders the hyperlink.

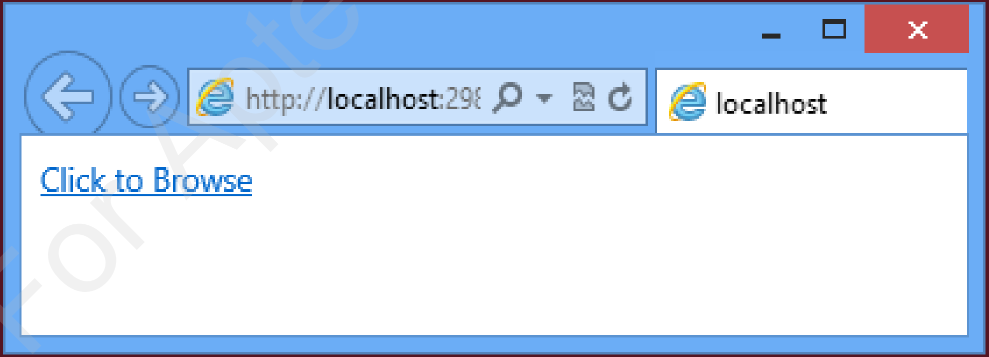
HTML Helper Methods 3-17

- Following code snippet shows using a `Html.ActionLink()` helper method:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
   @Html.ActionLink("Click to Browse", "Browse", "Home")
</body>
</html>
```

- In this code, the `Click to Browse` text will be displayed as a hyperlink.
- The `Browse` action method of the `Home` controller acts as the target of the hyperlink, as shown in the following figure:



HTML Helper Methods 4-17

- `Html.BeginForm()` helper method:
  - Allows you to mark the start of a form.
  - Coordinates with the routing engine to generate a URL.
  - Is responsible for producing the opening `<form>` tag.
- The general syntax of the `Html.BeginForm()` helper method is as follows:

Syntax:

```
@{Html.BeginForm(<action_method>,<controller_name>);}
```

where,

  - `action_method`: Is the name of the action method.
  - `controller_name`: Is the name of the controller class.
- Once you use the `Html.BeginForm()` helper method to start a form, you need to end a form using the `Html.EndForm()` helper method.

HTML Helper Methods 5-17

- Following code snippet shows using the `Html.BeginForm()` and `Html.EndForm()` helper methods:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
 @{Html.BeginForm("Browse","Home");}
    <p>Inside Form</p>
@{Html.EndForm();}
</body>
</html>
```

- In this code the `Html.BeginForm()` method specifies the `Browse` action of the `Home` controller as the target action method to which the form data will be submitted.
- You can also avoid using the `Html.EndForm()` helper method to explicitly close the form by using the `@using` statement before the `Html.BeginForm()` method.

HTML Helper Methods 6-17

- `Html.Label()` helper method:
  - Allows you to display a label in a form.
  - Enables attaching information to other input elements, such as text inputs, and increase the accessibility of your application.
- The general syntax of the `Html.Label()` helper method is as follows:

Syntax:

```
@Html.Label(<label_text>)
```

where,

- `label_text`: Is the name of the label.

HTML Helper Methods 6-17

- Following code snippet shows the `Html.Label()` method:

Code Snippet:

```
@Html.Label("name")
<!DOCTYPE html>
<html>
<body>
 @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @{Html.EndForm();}
</body>
</html>
```

- `Html.TextBox()` helper method:
  - Allows you to display an input tag.
  - Used to accept input from a user.

HTML Helper Methods 7-17

- The general syntax of the `Html.TextBox()` helper method is as follows:

Syntax:

```
@Html.TextBox("textbox_text”)
```

where,

- `textbox_text`: is the name of the text box

- Following code snippet shows using a `Html.TextBox()` method:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
 @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @Html.TextBox("textBox1")</br></br>
<input type="submit" value="Submit">
@{Html.EndForm();}
</body>
</html>
```

HTML Helper Methods 8-17

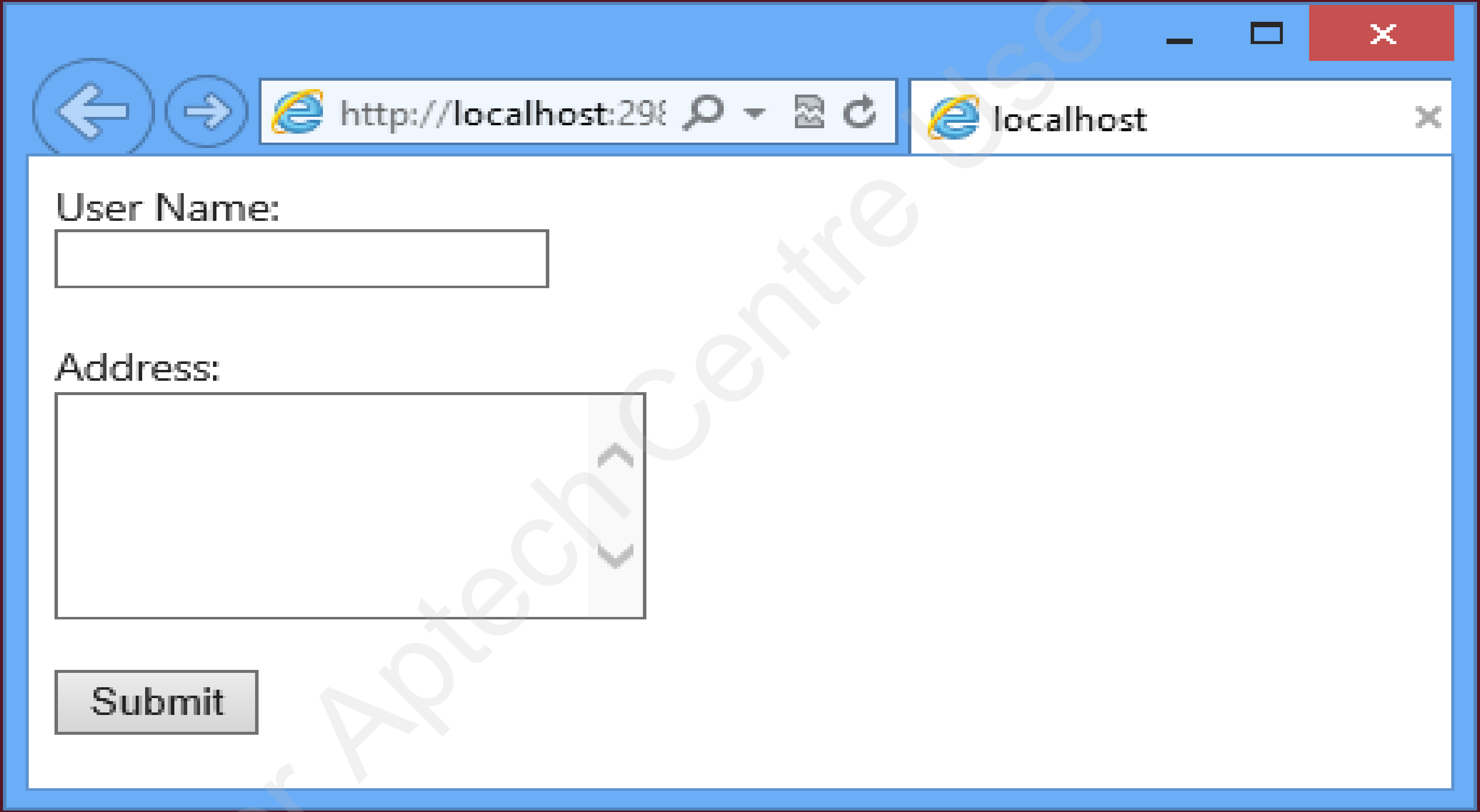
- `Html.TextArea()` helper method:
  - Allows you to display a `<textarea>` element for multi-line text entry.
  - Enables you to specify the number of columns and rows to be displayed in order to control the size of the text area.
- Following code snippet shows using a `Html.TextArea()` method:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
 @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @Html.TextBox("textBox1")</br></br>
 @Html.Label("Address:")</br>
 @Html.TextArea("textarea1")</br></br>
 <input type="submit" value="Submit">
@{Html.EndForm();}
</body>
</html>
```

HTML Helper Methods 9-17

- Following figure shows using the `Html.TextArea()` helper method:



HTML Helper Methods 10-17

- You can use the `Html.Password()` helper method to display a `password` field.
- Following code snippet shows using a `Html.Password()` method:

Code Snippet:

```
<!DOCTYPE html>
<html>
<body>
 @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @Html.TextBox("textBox1")</br></br>
 @Html.Label("Address:")</br> @Html.TextArea("textarea1")</br></br>
 @Html.Label("Password:")</br>@Html.Password("password")</br></br>
 <input type="submit" value="Submit">
@{Html.EndForm();}
</body>
</html>
```

HTML Helper Methods 11-17

- You can use the `Html.CheckBox()` helper method to display a check box that enables the user to select a true or false condition.
- Following code snippet shows using a `Html.CheckBox()` method:

Code Snippet:

```
<!DOCTYPE html>
<html><body>
 @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @Html.TextBox("textBox1")</br></br>
 @Html.Label("Address:")</br> @Html.TextArea("textarea1")</br></br>
 @Html.Label("Password:")</br>@Html.Password("password")</br></br>
 @Html.Label("I need updates on my mail:")
 @Html.CheckBox ("checkbox1")</br> </br>
 <input type="submit" value="Submit"> @{Html.EndForm();}
</body> </html>
```

- In this code, the `Html.CheckBox()` helper method renders a hidden input in addition to the check box input. The hidden input ensures that a value will be submitted, even if the user does not select the check box.

HTML Helper Methods 12-17

- `Html.DropDownList()` helper method:
  - Return a <`select`/> element that shows a list of possible options and also the current value for a field.
  - Allows selection of a single item.
- The general syntax of the `Html.DropDownList()` helper method is as follows:

Syntax:

```
@Html.DropDownList("myList", new SelectList(new [] {<value1>, <value2>,
< value3>}), "Choose")
```

where,

- `value1`, `value2`, and `value3` are the options available in the drop-down list.
- `Choose`: Is the value at the top of the list.

HTML Helper Methods 13-17

- Following code snippet shows using a `Html.DropDownList()` method:

Code Snippet:

```
<!DOCTYPE html>
<html><body>  @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @Html.TextBox("textBox1")</br></br>
 @Html.Label("Address:")</br> @Html.TextArea("textarea1")</br></br>
 @Html.Label("Password:")</br>@Html.Password("password")</br></br>
 @Html.Label("I need updates on my mail:")
 @Html.CheckBox("checkbox1")</br> </br>
 @Html.Label("Select your city:")
 @Html.DropDownList("myList", new SelectList(new [] {"New York",
"Philadelphia", "California"}), "Choose")</> </br></br>
 <input type="submit" value="Submit">
@{Html.EndForm();}
</body></html>
```

- In this code, the `Html.DropDownList()` method creates a drop-down list in a form with `myList` as its name and contains three values that a user can select from the drop-down list.

HTML Helper Methods 14-17

- The `Html.RadioButton()` helper method allows you to provide a range of possible options for a single value.
- The general syntax of the `Html.RadioButton()` helper method is as follows:

Syntax:

```
@Html.RadioButton("name", "value", isChecked)
```

where,

- `name`: Is the name of the radio button input element.
- `value`: Is the value associated with a particular radio button option.
- `isChecked`: Is a Boolean value that indicates whether the radio button option is selected or not.

HTML Helper Methods 15-17

- Following code snippet shows using a `Html.RadioButton()` method:

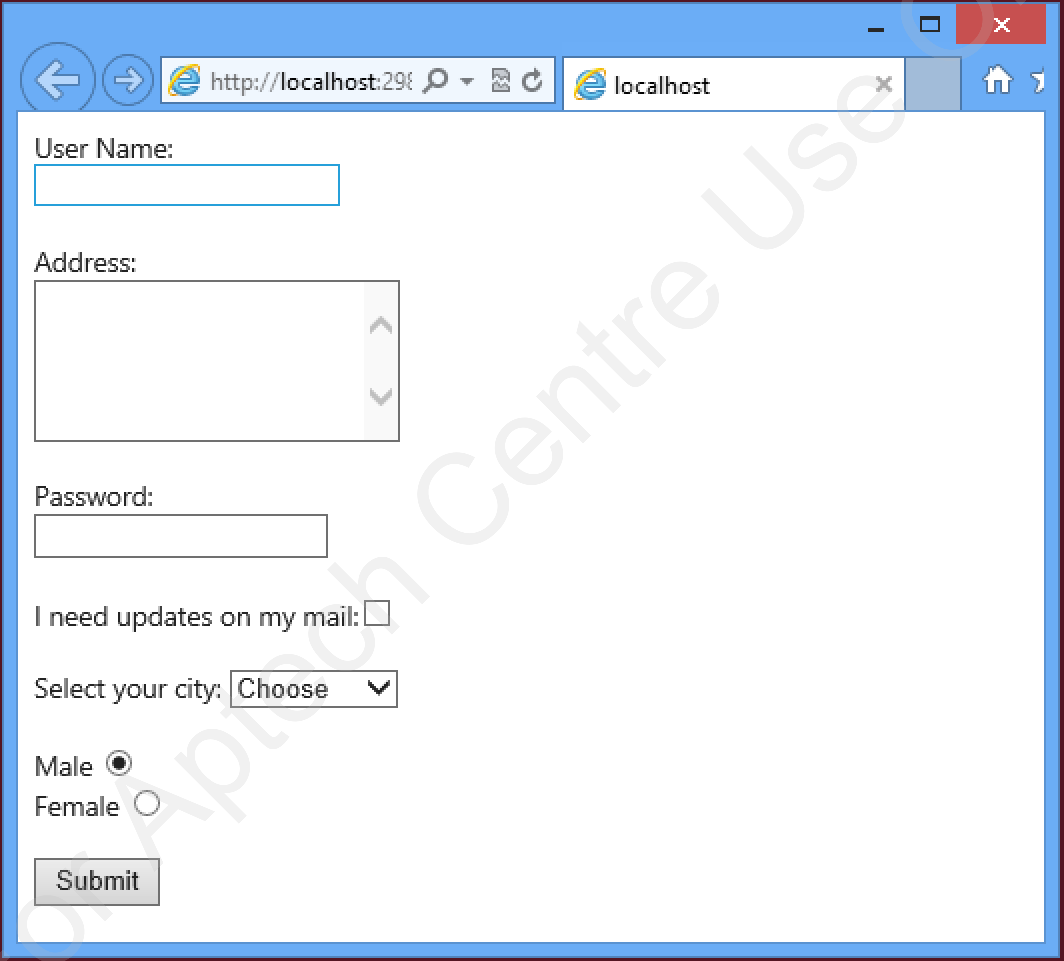
Code Snippet:

```
<!DOCTYPE html>
<html> <body>
 @{Html.BeginForm("Browse","Home");}
 @Html.Label("User Name:")</br>
 @Html.TextBox("textBox1")</br></br>
 @Html.Label("Address:")</br> @Html.TextArea("textarea1")</br></br>
 @Html.Label("Password:")</br>@Html.Password("password")</br></br>
 @Html.Label("I need updates on my
mail:")@Html.CheckBox("checkbox1")</br> </br>
 @Html.Label("Select your city:")  @Html.DropDownList("myList", new
SelectList(new [] {"New York", "Philadelphia", "California"}),
"Choose")</> </br></br>
 Male @Html.RadioButton("Gender", "Male", true)</br>
 Female @Html.RadioButton("Gender", "Female")</br> </br>
<input type="submit" value="Submit">
@{Html.EndForm();}</body> </html>
```

- In this code, the `Html.RadioButton()` helper methods is used to create two radio buttons to accept the gender of a user.

HTML Helper Methods 16-17

- Following figure shows using the `Html.RadioButton()` helper method:



User Name:

Address:

Password:

I need updates on my mail: ☐

Select your city: Choose

Male ●
Female ○

HTML Helper Methods 17-17

- The `Url.Action()` helper method generates a URL that targets a specified action method of a controller.
- The general syntax of the `Url.Action()` helper method is as follows:

Syntax:

```
@Url.Action(<action_name>, <controller_name>)
```

where,

- `action_name`: Is the name of the action method.
- `controller_name`: Is the name of the controller class.

- Following code snippet shows the `Url.Action()` method:

Code Snippet:

```
<!DOCTYPE html>
<html> <body>
    <a href='@Url.Action("Browse", "Home")'>Browse</a>
</body> </html>
```

- This code creates a hyperlink that targets the URL generated using the `Url.Action()` method. When a user clicks the hyperlink, the `Browse` action of the Home controller will be invoked.

Summary

- In an ASP.NET MVC application, views are used to display both static and dynamic content.
- View Engines are part of the MVC Framework that converts the code of a view into HTML markup that a browser can understand.
- You can use ViewData, ViewBag, and TempData to pass data from a controller to a view.
- In an ASP.NET MVC application, a partial view represents a sub-view of a main view.
- Razor is the syntax, based on the ASP.NET Framework that allows you to create views.
- The MVC Framework uses a view engine to convert the code of a view into HTML markup that a browser can understand.
- The MVC Framework provides built-in HTML helper methods that you can use to generate HTML markup and you can reuse it across the Web application.