Introduction

1. HTTP Protocol

1>HTTP is the foundation of data communication for the in the World Wide Web

2> HTTP functions as a request-response protocol in the client-server computing Model. In HTTP, a web browser for example, acts as a *client*, while an application running on a computer hosting a website functions as a server. The client submits an HTTP *request* message to the server. The server, which stores content, or provides *resources*, such as HTML files or performs other functions on behalf of the client, returns a response message to the client. A response contains completion status information about the request and may contain any content requested by the client in its message body.

It is a stateless protocol, you send request and it responds does not maintain any state. To maintain the state, we can use

1>Session Object in the server side

2>Cookies in the client side.

3>Hidden variables in the form

2 Static and Dynamic Pages

Static Web Pages

A **static web page** (sometimes called a **flat page)** is a web page that is delivered to the user exactly as stored, in contrast to dynamic web pages which are generated by a web application.

Consequently a static web page displays the same information for all users, from all contexts, subject to modern capabilities of a web server to negotiate content-type or language of the document where such versions are available and the server is configured to do so.

Static web pages are often HTML documents stored as files in the file system and made available by the web server over HTTP.

Advantages

- No programming skills are required to create a static page.
- Inherently publicly cacheable (ie. a cached copy can be shown to anyone).
- No particular hosting requirements are necessary.
- Can be viewed directly by a web browser without needing a web server or application server

Disadvantages

- Any personalization or interactivity has to run client-side (ie. in the browser), which is restricting.
- Maintaining large numbers of static pages as files can be impractical without automated tools.

Dynamic Web pages

A **dynamic web page** is a kind of web page that has been prepared with recent information, for each individual viewing. It changes with the time. For ex. News content, events. The content of the site changes when the user logs in and logs out. Page contents are displayed according to the location of the user. In facebook, you can see suggestions to add friends since it knows your friends and those are mostly friend of your friends. Facebook gets updated with recent feeds etc.

Dynamic web pages can be created in two ways

1>**Client-Side Scripting**

Using client-side scripting to change interface behaviors *within* a specific web page, in response to mouse or keyboard actions or at specified timing events. In this case the dynamic behavior occurs within the presentation.

Client-Side scripting languages like JavaScript or Actionscript, used for DHTML and Flash technologies respectively, are frequently used to play with the media types (sound, animations, changing text, etc.) of the presentation

The Client-side content is generated on the user's computer. The web browser retrieves a page from the server, then processes the code embedded in the page (often written in JavaScript) and displays the retrieved page's content to the user.

2>**Server-Side Scripting**

A program running on the web server (server-side scripting) is used to change the web content on various web pages, or to adjust the sequence of or reload of the web pages. Server responses may be determined by such conditions as data in a posted HTML form, parameters in the URL, the type of browser being used, the passage of time, or a database or server state.
Such web pages are often created with the help of server-side languages such as ASP, PHP, JSP etc.

3>**Combined Client Side and Server Side**

Ajax is a web development technique for dynamically interchanging content with the server-side, without reloading the web page. Google Maps is an example of a web application that uses Ajax techniques and database.

Disadvantages

Search engines work by creating indexes of published HTML web pages that were, initially, "static". With the advent of dynamic web pages, often created from a private database, the content is less visible. Unless this content is duplicated in some way (for example, as a series of extra static pages on the same site), a search may not find the information it is looking for. It is unreasonable to expect generalized web search engines to be able to access complex database structures, some of which in any case may be secure.

Introduction to ASP

Microsoft Active Server Pages (ASP) is a server-side scripting technology. ASP is a technology that Microsoft created to ease the development of interactive Web applications.

Using server side scripting language like ASP we can manage the content of any page and such dynamic code (or content) for the web browsers can be generated based on various conditions we set in our ASP program.

ASP engine finishes its job of processing the code and then send the codes to users browser. From this point on words till again the page request comes back to server, there is no control of ASP on the page. So we should not expect ASP to perform some tasks which are likely to happen at the client browser end. This will be clear when we discuss some of the task and where (client or server side) the task is to be completed and which script will take care of it.

ASP provides solutions for transaction processing and managing session state. Asp is one of the most successful language used in web development.

Benefits and Application of ASP

- Dynamically edit, change, or add any content of a Web page
- Respond to user queries or data submitted from HTML forms
- Access any data or databases and return the results to a browser
- Customize a Web page to make it more useful for individual users
- Provide security - since ASP code cannot be viewed from the browser
- Clever ASP programming can minimize the network traffic

Problems with traditional ASP

1. **Interpreted and Loosely-Typed Code**
   ASP scripting code is usually written in languages such as JScript or VBScript. The script-execution engine that Active Server Pages relies on interprets code line by line, every time the page is called. In addition, although variables are supported, they are all loosely typed as variants and bound to particular types only when the code is run. Both these factors impede performance, and late binding of types makes it harder to catch errors when you are writing code.

2. **Mixes layout (HTML) and logic (scripting code)**
   ASP files frequently combine script code with HTML. This results in ASP scripts that are lengthy, difficult to read, and switch frequently between code and HTML. The interspersion of HTML with ASP code is particularly problematic for larger web applications, where content must be kept separate from business logic.

3. **Limited Development and Debugging Tools**
   Microsoft Visual InterDev, Macromedia Visual UltraDev, and other tools have attempted to increase the productivity of ASP programmers by providing graphical development environments. However, these tools never achieved the ease of use or the level of acceptance achieved by Microsoft Windows application development tools, such as Visual Basic or Microsoft Access. ASP developers still rely heavily or exclusively on Notepad.

   Debugging is an unavoidable part of any software development process, and the debugging tools for ASP have been minimal. Most ASP programmers
   resort to embedding temporary Response. Write statements in their code to trace the progress of its execution.

4. **No real state management**
   Session state is only maintained if the client browser supports cookies. Session state information

can only be held by using the ASP Session object. And you have to implement additional code if you, for example, want to identify a user.

5. **Update files only when server is down**
   If your Web application makes use of components, copying new files to your application should only be done when the Web server is stopped. Otherwise it is like pulling the rug from under your application's feet, because the components may be in use (and locked) and must be registered.

6. **Obscure Configuration Settings**
   The configuration information for an ASP web application (such as session state and server timeouts) is stored in the IIS metabase. Because the metabase is stored in a proprietary format, it can only be modified on the server machine with utilities such as the Internet Service Manager. With limited support for programmatically manipulating or extracting these settings, it is often an arduous task to port an ASP application from one server to another.

Introduction to ASP.NET

ASP.NET Overview

Some point that gives the quick overview of ASP.NET.

- ASP.NET provides services to allow the creation, deployment, and execution of Web Applications and Web Services
- Like ASP, ASP.NET is a server-side technology
- Web Applications are built using Web Forms. ASP.NET comes with built-in Web Forms controls, which are responsible for generating the user interface. They mirror typical HTML widgets like text boxes or buttons. If these controls do not fit your needs, you are free to create your own user controls.
- Web Forms are designed to make building web-based applications as easy as building Visual Basic applications

Advantages of ASP.NET

1. **Separation of Code from HTML**
   To make a clean sweep, with ASP.NET you have the ability to completely separate layout and business logic. This makes it much easier for teams of programmers and designers to collaborate efficiently. This makes it much easier for teams of programmers and designers to collaborate efficiently.

2. **Support for compiled languages**
   Developer can use VB.NET and access features such as **strong typing** and **object-oriented programming**. Using compiled languages also means that ASP.NET pages do not suffer the performance penalties associated with interpreted code. ASP.NET pages are precompiled to byte-code and Just In Time (JIT) compiled when first requested. Subsequent requests are directed to the fully compiled code, which is cached until the source changes.

3. **Use services provided by the .NET Framework**
   The .NET Framework provides class libraries that can be used by your application. Some of the key classes help you with input/output, access to operating system services, data access, or even debugging. We will go into more detail on some of them in this module.

4. **Graphical Development Environment**
   Visual Studio .NET provides a very rich development environment for Web developers. You can drag and drop controls and set properties the way you do in Visual Basic 6. And you have full Intelligence support, not only for your code, but also for HTML and XML.

5. **State management**
   To refer to the problems mentioned before, ASP.NET provides solutions for session and application state management. State information can, for example, be kept in memory or stored in a database. It can be shared across Web farms, and state information can be recovered, even if the server fails or the connection breaks down.

6. **Update files while the server is running!**
   Components of your application can be updated while the server is online and clients are connected. The Framework will use the new files as soon as they are copied to the application. Removed or old files that are still in use are kept in memory until the clients have finished.

7. **XML-Based Configuration Files**
   Configuration settings in ASP.NET are stored in XML files that you can easily read and edit. You can also easily copy these to another server, along with the other files that comprise your application.

ASP.NET Architecture

ASP.NET is based on the fundamental architecture of .NET Framework.

Architecture is explained form bottom to top in the following discussion.

1. At the bottom of the Architecture is Common Language Runtime. NET Framework common language runtime resides on top of the operating system services. The common language runtime loads and executes code that targets the runtime. This code is therefore called managed code. The runtime gives you, for example, the ability for cross-language integration.

2. .NET Framework provides a rich set of class libraries. These include base classes, like networking and input/output classes, a data class library for data access, and classes for use by programming tools, such as debugging services. All of them are brought together by the Services Framework, which sits on top of the common language runtime.

3. ADO.NET is Microsoft ActiveX Data Object (ADO) model for the .NET Framework. ADO.NET is not simply the migration of the popular ADO model to the managed environment but a

completely new paradigm for data access and manipulation.

ADO.NET is intended specifically for developing web applications. This is evident from its two major design principles:

1. Disconnected Datasets In ADO.NET, almost all data manipulation is done outside the context of an open database connection.
2. Effortless Data Exchange with XMLDatasets can converse in the universal data format of the Web, namely XML.

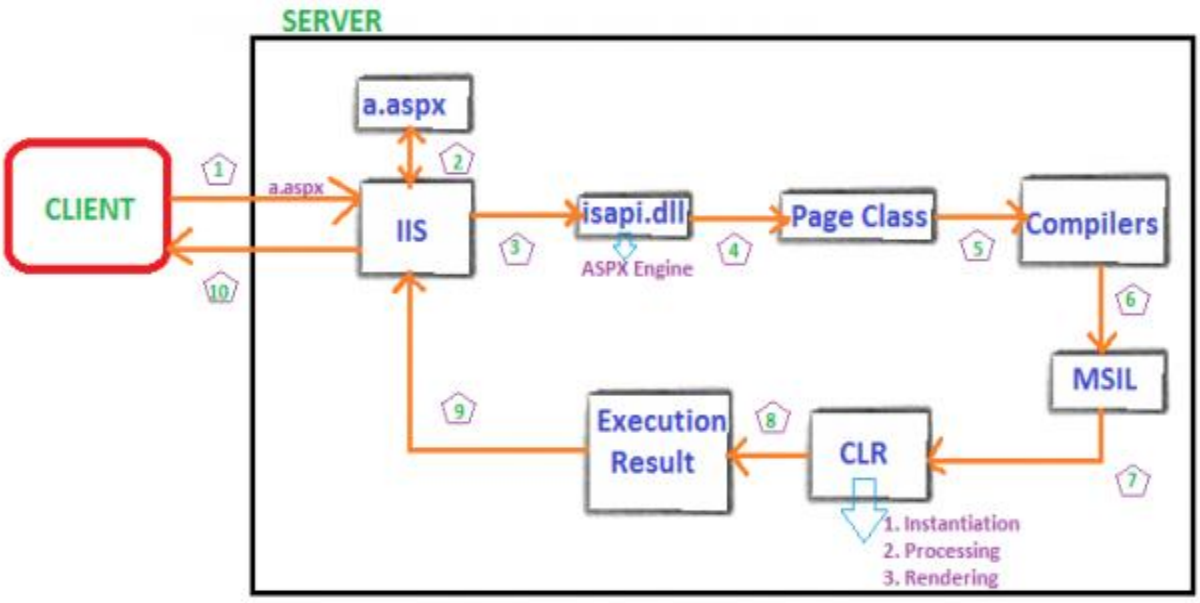
4. The 4th layer of the framework consists of the Windows application model and, in parallel, the Web application model.
   The Web application model-in the slide presented as ASP.NET-includes Web Forms and Web Services.
   ASP.NET comes with built-in Web Forms controls, which are responsible for generating the user interface. They mirror typical HTML widgets like text boxes or buttons. If these controls do not fit your needs, you are free to create your own user controls.

   Web Services brings you a model to bind different applications over the Internet. This model is based on existing infrastructure and applications and is therefore standard-based, simple, and adaptable.

   Web Services are software solutions delivered via Internet to any device. Today, that means Web browsers on computers, for the most part, but the device-agnostic design of .NET will eliminate this limitation.

5. One of the obvious themes of .NET is unification and interoperability between various programming languages. In order to achieve this; certain rules must be laid and all the languages must follow these rules. In other words we can not have languages running around creating their own extensions and their own fancy new data types. CLS(Common Language Specification) is the collection of the rules and constraints that every language (that seeks to achieve .NET compatibility) must follow.
6. The CLR and the .NET Frameworks in general, however, are designed in such a way that code written in one language can not only seamlessly be used by another language. Hence ASP.NET can be programmed in any of the .NET compatible language whether it is VB.NET, C#, Managed C++ or JScript.NET.

Execution Process



Compilation, when page is requested the first time

The first time a page is requested, the code is compiled. Compiling code in .NET means that a compiler in a first step emits Microsoft intermediate language (MSIL) and produces metadata if you compile your source code to managed code. In a following step MSIL has to be converted to native code.

Microsoft intermediate language (MSIL)

Microsoft intermediate language is code in an assembly language like style. It is CPU independent and therefore can be efficiently converted to native code.

The conversion in turn can be CPU-specific and optimized. The intermediate language provides a hardware abstraction layer.

MSIL is executed by the common language runtime.

Common language runtime

The common language runtime contains just-in-time (JIT) compilers to convert the MSIL into native code. This is done on the same computer architecture that the code should run on.

The runtime manages the code when it is compiled into MSIL the code is therefore called managed code.

IIS(Internet Information Server)

Internet Information Services (**IIS**) – formerly called **Internet Information Server** – is a web server application and set of feature extension modules created by Microsoft for use with Microsoft Windows. It is the most used web server after Apache HTTP Server. IIS 7.5 supports HTTP, HTTPS, FTP, FTPS, SMTP .

Hypertext Transfer Protocol Secure (**HTTPS**) is a combination of the Hypertext Transfer Protocol (HTTP) with SSL/TLS protocol to provide encrypted communication and secure identification of a

network web server. HTTPS connections are often used for payment transactions on the World Wide Web and for sensitive transactions in corporate information systems

It is an integral part of Windows Server family of products, as well as certain editions of Windows XP, Windows Vista and Windows 7. IIS is not turned on by default when Windows is installed.

IIS Features

The architecture of IIS 7 is modular. Modules, also called extensions, can be added or removed individually so that only modules required for specific functionality have to be installed. IIS 7 includes native modules as part of the full installation. These modules are individual features that the server uses to process requests and include the following:

Native modules – Come with the IIS installation

- **HTTP modules** – Used to perform tasks specific to HTTP in the request-processing pipeline, such as responding to information and **inquiries sent in client headers, returning HTTP errors, and redirecting requests.**

- **Security modules** – Used to perform tasks related to security in the request-processing pipeline, such as specifying authentication schemes, performing URL authorization, and filtering requests. Prohibits file listing.

- **Content modules** – Used to perform tasks related to content in the request-processing pipeline, such as processing requests for static files, returning a default page when a client does not specify a resource in a request, and listing the contents of a directory. We can set the default file names in IIS.

- **Compression modules** – Used to perform tasks related to compression in the request-processing pipeline, such as compressing responses, applying Gzip compression transfer coding to responses, and performing pre-compression of static content.

- **Caching modules** – Used to perform tasks related to caching in the request-processing pipeline, such as storing processed information in memory on the server and using cached content in subsequent requests for the same resource.

- **Logging and Diagnostics modules** – Used to perform tasks related to logging and diagnostics in the request-processing pipeline, such as passing information and processing status to HTTP.sys for logging, reporting events, and tracking requests currently executing in worker processes.

Sites, Applications, and Virtual Directories in IIS

IIS formalizes the concepts of sites, applications, and virtual directories. Virtual directories and applications are now separate objects, and they exist in a hierarchical relationship in the IIS configuration schema. Briefly, a site contains one or more applications, an application contains one or more virtual directories, and a virtual directory maps to a physical directory on a computer.

As in IIS 6.0, a site contains all the content, both static and dynamic, that is associated with that site. However, each site must contain at least one application, which is named the root application. And each

application (including the root application) must contain at least one virtual directory, which is named the root virtual directory. These objects work together to form the site.

The following sections explain sites, applications, virtual directories, and their related configurations in more detail.

Sites

A site is a container for applications and virtual directories, and you can access it through one or more **unique bindings.**

The binding includes two attributes important for communication: the ***binding protocol*** **and the** ***binding information***. The binding protocol defines the protocol over which communication between the server and client occurs. The binding information defines the information that is used to access the site. For example, the binding protocol of a Web site can be either HTTP or HTTPS, and the binding information is the combination of IP address, port, and optional host header.

A site may contain more than one binding if the site requires different protocols or binding information. In earlier versions of IIS, only the HTTP and HTTPS protocols were supported. For example, a Web site might have had both an HTTP binding and an HTTPS binding when sections of the site required secure communication over HTTPS(for example when you do transactions online, credit card information sharing) .

Applications

An application is a group of files that delivers content or provides services over protocols, such as HTTP. When you create an application in IIS, the application's path becomes part of the site's URL.

In IIS 7, each site must have an application which is named the root application, or default application. However, a site can have more than one application. For example, you might have an online commerce Web site that has several applications, such as a shopping cart application that lets users gather items during shopping and a login application that allows users to recall saved payment information when they make a purchase.

Virtual Directories

A virtual directory is a directory name (also referred to as path) that you specify in IIS and map to a physical directory on a local or remote server. The directory name then becomes part of the application's URL, and users can request the URL from a browser to access content in the physical directory, such as a Web page or a list of additional directories and files. If you specify a different name for the virtual directory than the physical directory, it is more difficult for users to discover the actual physical file structure on your server because the URL does not map directly to the root of the site.

In IIS 7, each application must have a virtual directory, which is named the root virtual directory, and which maps the application to the physical directory that contains the application's content. However, an application can have more than one virtual directory. For example, you might use a virtual directory when you want your application to include images from another location in the file system, but you do not want to move the image files into the physical directory that is mapped to the application's root virtual directory.

By default, IIS uses configuration from Web.config files in the physical directory to which the virtual directory is mapped, as well as in any child directories in that physical directory.

Optionally, when you need to specify credentials and a method to access the virtual directory, you can specify values for the **username**, **password**, and **logonMethod** attributes.

Response Object

The ASP Response object is used to send output to the user from the server. Its collections, properties, and methods are described below:

Collections

Cookies

The Cookies collection is used to set or get cookie values. If the cookie does not exist, it will be created, and take the value that is specified.

Syntax

Response.Cookies(name)[(key)|.attribute]=value

variablename=Request.Cookies(name)[(key)|.attribute]

Parameter	Description
Name	Required. The name of the cookie
Value	Required for the Response.Cookies command. The value of the cookie
Attribute	Optional. Specifies information about the cookie. Can be one of the following parameters: • Domain - Write-only. The cookie is sent only to requests to this domain • Expires - Write-only. The date when the cookie expires. If no date is specified, the cookie will expire when the session ends • HasKeys - Read-only. Specifies whether the cookie has keys (This is the only attribute that can be used with the Request.Cookies command) • Path - Write-only. If set, the cookie is sent only to requests to this path. If not set, the application path is used • Secure - Write-only. Indicates if the cookie is secure
Key	Optional. Specifies the key to where the value is assigned

Examples

The "Response.Cookies" command is used to create a cookie or to set a cookie value:

```
<%
Response.Cookies("firstname")="Alex"
%>
```

In the code above, we have created a cookie named "firstname" and assigned the value "Alex" to it.

It is also possible to assign some attributes to a cookie, like setting a date when a cookie should expire:

```
<%
Response.Cookies("firstname")="Alex"
Response.Cookies("firstname").Expires=#May 10,2002#
%>
```
Now the cookie named "firstname" has the value of "Alex", and it will expire from the user's computer at May 10, 2002.
The "Request.Cookies" command is used to get a cookie value.
In the example below, we retrieve the value of the cookie "firstname" and display it on a page:
```
<%
fname=Request.Cookies("firstname")
response.write("Firstname=" & fname)
%>
```

Output:
Firstname=Alex

A cookie can also contain a collection of multiple values. We say that the cookie has Keys.

In the example below, we will create a cookie-collection named "user". The "user" cookie has Keys that contains information about a user:

```
<%
Response.Cookies("user")("firstname")="John"
Response.Cookies("user")("lastname")="Smith"
Response.Cookies("user")("country")="Norway"
Response.Cookies("user")("age")="25"
%>
```

The code below reads all the cookies your server has sent to a user. Note that the code checks if a cookie has Keys with the HasKeys property:

```
<html>
<body>

<%
dim x,y

for each x in Request.Cookies
  response.write("<p>")
  if Request.Cookies(x).HasKeys then
    for each y in Request.Cookies(x)
      response.write(x & ":" & y & "=" & Request.Cookies(x)(y))
      response.write("<br /")
    next
  else
    Response.Write(x & "=" & Request.Cookies(x) & "<br />")
  end if
  response.write "</p>"
next
%>
```

```
</body>
</html>
%>
```

Output:

```
firstname=Alex

user:firstname=John
user:lastname=Smith
user:
country=Norway
user:
age=25
```

Properties

1. Buffer

The Buffer property specifies whether to buffer the output or not. When the output is buffered, the server will hold back the response to the browser until all of the server scripts have been processed, or until the script calls the Flush or End method.

Note: If this property is set, it should be before the <html> tag in the .asp file

Syntax

response.Buffer[=flag]

Parameter	Description
Flag	A boolean value that specifies whether to buffer the page output or not. False indicates no buffering. The server will send the output as it is processed. False is default for IIS version 4.0 (and earlier). Default for IIS version 5.0 (and later) is true. True indicates buffering. The server will not send output until all of the scripts on the page have been processed, or until the Flush or End method has been called.

Examples

Example 1

In this example, there will be no output sent to the browser before the loop is finished. If buffer was set to False, then it would write a line to the browser every time it went through the loop.

```
<%response.Buffer=true%>
<html>
<body>
<%
for i=1 to 100
  response.write(i & "<br />")
next
%>
</body>
</html>
```

Example 2

```
<%response.Buffer=true%>
<html>
<body>
<p>I write some text, but I will control when
the text will be sent to the browser.</p>
<p>The text is not sent yet. I hold it back!</p>
<p>OK, let it go!</p>
<%response.Flush%>
</body>
</html>
```

Example 3

```
<%response.Buffer=true%>
<html>
<body>
<p>This is some text I want to send to the user.</p>
<p>No, I changed my mind. I want to clear the text.</p>
<%response.Clear%>
</body>
</html>
```

2. ContentType

The ContentType property sets the HTTP content type for the response object.

Syntax

```
response.ContentType[=contenttype]
```

Parameter	Description
Contenttype	A string describing the content type. For a full list of content types, see your browser documentation or the HTTP specification.

Examples

If an ASP page has no ContentType property set, the default content-type header would be:

content-type:text/html

Some other common ContentType values:

```
<%response.ContentType="text/HTML"%>
<%response.ContentType="image/GIF"%>
<%response.ContentType="image/JPEG"%>
<%response.ContentType="text/plain"%>
```

This example will open an Excel spreadsheet in a browser (if the user has Excel installed):

```
<%response.ContentType="application/vnd.ms-excel"%>
<html>
<body>
<table>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</table>
</body>
</html>
```

3. Expires

The Expires property sets how long (in minutes) a page will be cached on a browser before it expires. If a user returns to the same page before it expires, the cached version is displayed.

Syntax

response.Expires[=number]

Parameter	Description
Number	The time in minutes before the page expires

Examples

Example 1

The following code indicates that the page will never be cached:

<%response.Expires=-1%>

Example 2

The following code indicates that the page will expire after 1440 minutes (24 hours):

<%response.Expires=1440%>

4. ExpiresAbsolute

he ExpiresAbsolute property sets a date and time when a cached page on a browser will expire. If a user returns to the same page before this date/time, the cached version is displayed.

Syntax
response.ExpiresAbsolute[=[date][time]]

Parameter	Description
Date	Specifies the date on which the page will expire. If this parameter is not specified, the page will expire at the specified time on the day that the script is run.
Time	Specifies the time at which the page will expire. If this parameter is not specified, the page will expire at midnight of the specified day.

Examples

The following code indicates that the page will expire at 4:00 PM on October 11, 2012:

<%response.ExpiresAbsolute=#October 11,2012 16:00:00#%>

5. Status

The Status property specifies the value of the status line returned by the server.

Tip: Use this property to modify the status line returned by the server.

Syntax
response.Status=statusdescription

Parameter	Description
statusdescription	A three-digit number and a description of that code, like 404 Not Found. **Note:** Status values are defined in the HTTP specification.

Examples
```
<%
ip=request.ServerVariables("REMOTE_ADDR")
if ip<>"194.248.333.500" then
  response.Status="401 Unauthorized"
  response.Write(response.Status)
  response.End
end if
%>
```

Methods

1.Clear

The Clear method clears any buffered HTML output.

Note: This method does not clear the response headers, only the response body.

Note: If response.Buffer is false, this method will cause a run-time error.

Syntax
```
response.Clear
```

Examples
```
<%
response.Buffer=true
%>
<html>
<body>
<p>This is some text I want to send to the user.</p>
<p>No, I changed my mind. I want to clear the text.</p>
<%
response.Clear
%>
</body>
</html>
```

Output:

(nothing)

2.End

The End method stops processing a script, and returns the current result.

Note: This method will flush the buffer if Response.Buffer has been set to true. If you do not want to return any output to the user, you should call Response.Clear first.

Syntax
Response.End

Examples
```
<html>
<body>
<p>I am writing some text. This text will never be
<%
Response.End
%>
finished! It's too late to write more!</p>
</body>
</html>
```

Output:

I am writing some text. This text will never be

3. Flush

The Flush method sends buffered HTML output immediately.

Note: If response.Buffer is false, this method will cause a run-time error.

Syntax
Response.Flush

Example
```
<%
Response.Buffer=true
%>
<html>
<body>
<p>I write some text, but I will control when the
text will be sent to the browser.</p>
<p>The text is not sent yet. I hold it back!</p>
<p>OK, let it go!</p>
<%
Response.Flush
%>
</body>
</html>
```

Output:

I write some text, but I will control when the
text will be sent to the browser.

The text is not sent yet. I hold it back!

OK, let it go!

4. Redirect

The Redirect method redirects the user to a different URL.

Syntax
Response.Redirect URL

Parameter	Description
URL	Required. The URL that the user (browser) is redirected to

Examples
<%
Response.Redirect "http://cloudfactory.com"
%>

5. Write

The Write method writes a specified string to the output.

Syntax

Response.Write variant/variable

Parameter	Description
Variant	Required. The data to write

Examples

Example 1
<%
Response.Write "Hello World"
%>

Output:

Hello World

Example 2
<%
name="John"
Response.Write(name)
%>

Output:

John

Example 3
<%
Response.Write("Hello
World")
%>

Output:

Hello
World

Request Object

When a browser asks for a page from a server, it is called a request. The Request object is used to get information from a visitor. Its collections, properties, and methods are described below:

Collections

1.**Cookies**(Refer to Response Object)

2.**Form**

The Form collection is used to retrieve the values of form elements from a form that uses the POST method.

Syntax

Request.Form(element)[(index)|.Count]

Parameter	Description
Element	Required. The name of the form element from which the collection is to retrieve values
Index	Optional. Specifies one of multiple values for a parameter. From 1 to Request.Form(parameter).Count.

Examples

Example 1

You can loop through all the values in a form request. If a user filled out a form by specifying two values - Blue and Green - for the color element, you could retrieve those values like this:

```
<% for i=1 to Request.Form("color").Count
  Response.Write(Request.Form("color")(i) & "<br />")
next
%>
```

Output:

Blue
Green

Example 2

Consider the following form:

```
<form action="submit.asp" method="post">
<p>First name: <input name="firstname"></p>
<p>Last name: <input name="lastname"></p>
<p>Your favorite color:
<select name="color">
<option>Blue</option>
<option>Green</option>
<option>Red</option>
<option>Yellow</option>
<option>Pink</option>
</select>
</p>
<p><input type="submit"></p>
</form>
```

The following request might be sent:

firstname=John&lastname=Dove&color=Red

Now we can use the information from the form in a script:

```
Hi, <%=Request.Form("firstname")%>.
Your favorite color is <%=Request.Form("color")%>.
```

Output:

Hi, John. Your favorite color is Red.

If you do not specify any element to display, like this:

Form data is: <%=Request.Form%>

the output would look like this:

Form data is: firstname=John&lastname=Dove&color=Red

3.QueryString:

The QueryString collection is used to retrieve the variable values in the HTTP query string.

The HTTP query string is specified by the values following the question mark (?), like this:

Link with a query string

The line above generates a variable named txt with the value "this is a query string test".

Query strings are also generated by form submission, or by a user typing a query into the address bar of the browser.

Note: If you want to send large amounts of data (beyond 100 kb) the Request.QueryString cannot be used.

Syntax

Request.QueryString(variable)[(index)|.Count]

Parameter	Description
Variable	Required. The name of the variable in the HTTP query string to retrieve
Index	Optional. Specifies one of multiple values for a variable. From 1 to Request.QueryString(variable).Count

Examples

Example 1

To loop through all the n variable values in a Query String:

The following request is sent:

http://www.w3schools.com/test/names.asp?n=John&n=Susan

and names.asp contains the following script:

```
<%
for i=1 to Request.QueryString("n").Count
  Response.Write(Request.QueryString("n")(i) & "<br />")
next
%>
```

The file names.asp would display the following:

John
Susan

Example 2

The following string might be sent:

http://cloudfactory.com/names.asp?name=John&age=30

this results in the following QUERY_STRING value:

name=John&age=30

Now we can use the information in a script:

```
Hi, <%=Request.QueryString("name")%>.
Your age is <%= Request.QueryString("age")%>.
```

Output:

Hi, John. Your age is 30.

If you do not specify any variable values to display, like this:

```
Query string is: <%=Request.QueryString%>
```

the output would look like this:

Query string is: name=John&age=30

4. ServerVariables:

The ServerVariables collection is used to retrieve the server variable values.

Syntax

Request.ServerVariables (server_variable)

Parameter	Description

server_variable	Required. The name of the server variable to retrieve

Server Variables

Variable	Description
ALL_HTTP	Returns all HTTP headers sent by the client. Always prefixed with HTTP_ and capitalized
ALL_RAW	Returns all headers in raw form
APPL_MD_PATH	Returns the meta base path for the application for the ISAPI DLL
APPL_PHYSICAL_PATH	Returns the physical path corresponding to the meta base path
AUTH_PASSWORD	Returns the value entered in the client's authentication dialog
AUTH_TYPE	The authentication method that the server uses to validate users
AUTH_USER	Returns the raw authenticated user name
CERT_COOKIE	Returns the unique ID for client certificate as a string
CERT_FLAGS	bit0 is set to 1 if the client certificate is present and bit1 is set to 1 if the cCertification authority of the client certificate is not valid
CERT_ISSUER	Returns the issuer field of the client certificate
CERT_KEYSIZE	Returns the number of bits in Secure Sockets Layer connection key size
CERT_SECRETKEYSIZE	Returns the number of bits in server certificate private key
CERT_SERIALNUMBER	Returns the serial number field of the client certificate
CERT_SERVER_ISSUER	Returns the issuer field of the server certificate
CERT_SERVER_SUBJECT	Returns the subject field of the server certificate
CERT_SUBJECT	Returns the subject field of the client certificate
CONTENT_LENGTH	Returns the length of the content as sent by the client
CONTENT_TYPE	Returns the data type of the content
GATEWAY_INTERFACE	Returns the revision of the CGI specification used by the server

HTTP_<*HeaderName*>	Returns the value stored in the header *HeaderName*
HTTP_ACCEPT	Returns the value of the Accept header
HTTP_ACCEPT_LANGUAGE	Returns a string describing the language to use for displaying content
HTTP_COOKIE	Returns the cookie string included with the request
HTTP_REFERER	Returns a string containing the URL of the page that referred the request to the current page using an <a> tag. If the page is redirected, HTTP_REFERER is empty
HTTP_USER_AGENT	Returns a string describing the browser that sent the request
HTTPS	Returns ON if the request came in through secure channel or OFF if the request came in through a non-secure channel
HTTPS_KEYSIZE	Returns the number of bits in Secure Sockets Layer connection key size
HTTPS_SECRETKEYSIZE	Returns the number of bits in server certificate private key
HTTPS_SERVER_ISSUER	Returns the issuer field of the server certificate
HTTPS_SERVER_SUBJECT	Returns the subject field of the server certificate
INSTANCE_ID	The ID for the IIS instance in text format
INSTANCE_META_PATH	The meta base path for the instance of IIS that responds to the request
LOCAL_ADDR	Returns the server address on which the request came in
LOGON_USER	Returns the Windows account that the user is logged into
PATH_INFO	Returns extra path information as given by the client
PATH_TRANSLATED	A translated version of PATH_INFO that takes the path and performs any necessary virtual-to-physical mapping
QUERY_STRING	Returns the query information stored in the string following the question mark (?) in the HTTP request
REMOTE_ADDR	Returns the IP address of the remote host making the request
REMOTE_HOST	Returns the name of the host making the request

REMOTE_USER	Returns an unmapped user-name string sent in by the user
REQUEST_METHOD	Returns the method used to make the request
SCRIPT_NAME	Returns a virtual path to the script being executed
SERVER_NAME	Returns the server's host name, DNS alias, or IP address as it would appear in self-referencing URLs
SERVER_PORT	Returns the port number to which the request was sent
SERVER_PORT_SECURE	Returns a string that contains 0 or 1. If the request is being handled on the secure port, it will be 1. Otherwise, it will be 0
SERVER_PROTOCOL	Returns the name and revision of the request information protocol
SERVER_SOFTWARE	Returns the name and version of the server software that answers the request and runs the gateway
URL	Returns the base portion of the URL

Examples

You can loop through all of the server variables like this:

```
<%
for each x in Request.ServerVariables
  response.write(x & "<br />")
next
%>
```

The following example demonstrates how to find out the visitor's browser type, IP address, and more:

```
<html>
<body>
<p>
<b>You are browsing this site with:</b>
<%Response.Write(Request.ServerVariables("http_user_agent"))%>
</p>
<p>
<b>Your IP address is:</b>
<%Response.Write(Request.ServerVariables("remote_addr"))%>
</p>
<p>
<b>The DNS lookup of the IP address is:</b>
<%Response.Write(Request.ServerVariables("remote_host"))%>
</p>
<p>
```

```
<b>The method used to call the page:</b>
<%Response.Write(Request.ServerVariables("request_method"))%>
</p>
<p>
<b>The server's domain name:</b>
<%Response.Write(Request.ServerVariables("server_name"))%>
</p>
<p>
<b>The server's port:</b>
<%Response.Write(Request.ServerVariables("server_port"))%>
</p>
<p>
<b>The server's software:</b>
<%Response.Write(Request.ServerVariables("server_software"))%>
</p>
</body>
</html>
```

Properties

TotalBytes

The TotalBytes property is a read-only property that returns the total number of bytes the client sent in the body of the request.

Syntax
varbytes=Request.Totalbytes

Example

The following code sets the variable a equal to the total number of bytes sent in the body of the request:

```
<%
dim a
a=Request.TotalBytes
%>
```

Server Object

The ASP Server object is used to access properties and methods on the server. Its properties and methods are described below:

Properties

ScriptTimeOut

The ScriptTimeout property sets or returns the maximum number of seconds a script can run before it is terminated.

Syntax

Server.ScriptTimeout[=NumSeconds]

Parameter	Description
NumSeconds	The maximum number of seconds a script can run before the server terminates it. Default is 90 seconds

Examples

Example 1

Set the script timeout:

```
<%
Server.ScriptTimeout=200
%>
```

Example 2

Retrieve the current value of the ScriptTimeout property:

```
<%
response.write(Server.ScriptTimeout)
%>
```

Methods

1.CreateObject

The CreateObject method creates an instance of an object.

Note: Objects created with this method have page scope. They are destroyed when the server are finished processing the current ASP page. To create an object with session or application scope, you can either use the <object> tag in the Global.asa file, or store the object in a session or application variable.

Syntax
Server.CreateObject(progID)

Part	Description
progID	Required. The type of object to create

Example 1

This example creates an instance of the server component MSWC.AdRotator:

```
<%
Set adrot=Server.CreateObject("MSWC.AdRotator")
%>
```

Example 2

An object stored in a session variable is destroyed when the session ends. However, you can also destroy the object by setting the variable to Nothing or to a new value:

```
<%
Session("ad")=Nothing
%>
```

or

```
<%
Session("ad")="a new value"
%>
```

Example 3

You cannot create an instance of an object with the same name as a built-in object:

```
<%
Set Application=Server.CreateObject("Application")
%>
```

2. Execute

The Execute method executes an ASP file from inside another ASP file. After executing the called .asp file, the control is returned to the original .asp file.

Syntax
Server.Execute(path)

Parameter	Description
Path	Required. The location of the ASP file to execute

Example
File1.asp:

```
<%
response.write("I am in File 1!<br />")
```

```
Server.Execute("file2.asp")
response.write("I am back in File 1!")
%>
```

File2.asp:

```
<%
response.write("I am in File 2!<br />")
%>
```

Output:

```
I am in File 1!
I am in File 2!
I am back in File 1!
```

Also look at the Server.Transfer method to see the difference between the Server.Execute and Server.Transfer methods.

3.GetLastError

The GetLastError method returns an ASPError object that describes the error condition that occurred.

By default, a Web site uses the file \iishelp\common\500-100.asp for processing ASP errors. You can either use this file, or create your own. If you want to change the ASP file for processing the 500;100 custom errors you can use the IIS snap-in.

Note: A 500;100 custom error will be generated if IIS encounters an error while processing either an ASP file or the application's Global.asa file.

Note: This method is available only before the ASP file has sent any content to the browser.

Syntax
```
Server.GetLastError()
```

Examples

Example 1

In the example an error will occur when IIS tries to include the file, because the include statement is missing the file parameter:

```
<!--#include f="header.inc" -->
<%
response.write("sometext")
%>
```

Example 2

In this example an error will occur when compiling the script, because the "next" keyword is missing:

```
<%
dim i
for i=1 to 10
  ........
nxt
%>
```

Example 3

In this example an error will occur because the script attempts to divide by 0:

```
<%
dim i,tot,j
i=0
tot=0
j=0

for i=1 to 10
  tot=tot+1
next

tot=tot/j
%>
```

4. MapPath

The MapPath method maps a specified path to a physical path.

Note: This method cannot be used in Session.OnEnd and Application.OnEnd.

Syntax

Server.MapPath(path)

Parameter	Description
Path	Required. A relative or virtual path to map to a physical path. If this parameter starts with / or \, it returns a path as if this parameter is a full virtual path. If this parameter doesn't start with / or \, it returns a path relative to the directory of the .asp file being processed

Examples

Example 1

For the example below, the file "test.asp" is located in C:\Inetpub\Wwwroot\Script.

The file "test.asp" (located in C:\Inetpub\Wwwroot\Script) contains the following code:

```
<%
response.write(Server.MapPath("test.asp") & "<br />")
response.write(Server.MapPath("script/test.asp") & "<br />")
response.write(Server.MapPath("/script/test.asp") & "<br />")
response.write(Server.MapPath("\script") & "<br />")
response.write(Server.MapPath("/") & "<br />")
response.write(Server.MapPath("\") & "<br />")
%>
```

Output:

```
c:\inetpub\wwwroot\script\test.asp
c:\inetpub\wwwroot\script\script\test.asp
c:\inetpub\wwwroot\script\test.asp
c:\inetpub\wwwroot\script
c:\inetpub\wwwroot
c:\inetpub\wwwroot
```

Example 2

How to use a relative path to return the relative physical path to the page that is being viewed in the browser:

```
<%
response.write(Server.MapPath("../"))
%>
```

or

```
<%
response.write(Server.MapPath("..\"))
%>
```

5. Transfer

The Transfer method sends (transfers) all the state information (all application/session variables and all items in the request collections) created in one ASP file to a second ASP file.

When the second ASP page completes its tasks, it will NOT return to the first ASP page (like the Execute method).

Note: The Transfer method is an efficient alternate for the Response.Redirect. A redirect forces the Web server to handle an extra request while the Server.Transfer method transfers execution to a different ASP page on the server, and avoids the extra round trip.

Syntax
Server.Transfer(path)

Parameter	Description
Path	Required. The location of the ASP file to which control should be transferred

Example
File1.asp:

```
<%
response.write("Line 1 in File 1<br />")
Server.Transfer("file2.asp")
response.write("Line 2 in File 1<br />")
%>
```

File2.asp:

```
<%
response.write("Line 1 in File 2<br />")
response.write("Line 2 in File 2<br />")
%>
```

Output:

Line 1 in File 1
Line 1 in File 2
Line 2 in File 2

Application Object

An application on the Web may consists of several ASP files that work together to perform some purpose. The Application object is used to tie these files together.

The Application object is used to store and access variables from any page, just like the Session object. The difference is that ALL users share ONE Application object (with Sessions there is ONE Session object for EACH user).

The Application object holds information that will be used by many pages in the application (like database connection information). The information can be accessed from any page. The information can also be changed in one place, and the changes will automatically be reflected on all pages.

The Application object is initialized by IIS when the first .asp page from within the given virtual directory is requested. It remains in the server's memory until either the web service is stopped or the application is explicitly unloaded from the web server

The Application object's collections, methods, and events are described below:

Collections

1.Contents

The Contents collection contains all the items appended to the application/session through a script command.

Tip: To remove items from the Contents collection, use the Remove and RemoveAll methods.

Syntax

Application.Contents(Key)

Parameter	Description
Key	Required. The name of the item to retrieve

Example 1

Notice that both name and objtest would be appended to the Contents collection:

```
<%
Application("name")="Harry"
Set Application("objtest")=Server.CreateObject("ADODB.Connection")
%>
```

Example 2

To loop through the Contents collection:

```
<%
for each x in Application.Contents
  Response.Write(x & "=" & Application.Contents(x) & "<br />")
next
%>
```

or:

```
<%
For i=1 to Application.Contents.Count
  Response.Write(i & "=" & Application.Contents(i) & "<br />")
Next
%>
```

Example 3

```
<%
Application("date")="2001/05/05"
Application("author")="Michael"

for each x in Application.Contents
  Response.Write(x & "=" & Application.Contents(x) & "<br />")
```

```
next
%>
```

Output:

```
date=2001/05/05
author= Michael
```

2.StaticObjects

The StaticObjects collection contains all the objects appended to the application/session with the HTML <object> tag.

Syntax

Application.StaticObjects(Key)

Parameter	Description
key	Required. The name of the item to retrieve

Example 1

To loop through the StaticObjects collection:

```
<%
for each x in Application.StaticObjects
  Response.Write(x & "<br />")
next
%>
```

Example 2

In Global.asa:

```
<object runat="server" scope="application"
id="MsgBoard" progid="msgboard.MsgBoard">
</object>

<object runat="server" scope="application"
id="AdRot" progid="MSWC.AdRotator">
</object>
```

In an ASP file:

```
<%
for each x in Application.StaticObjects
  Response.Write(x & "<br />")
next
%>
```

Output:

MsgBoard
AdRot

Methods

1. Contents.Remove

The Contents.Remove method deletes an item from the Contents collection.

Syntax

Application.Contents.Remove(name|index)

Parameter	Description
name	The name of the item to remove
index	The index of the item to remove

Example 1

```
<%
Application("test1")=("First test")
Application("test2")=("Second test")
Application("test3")=("Third test")

Application.Contents.Remove("test2")

for each x in Application.Contents
  Response.Write(x & "=" & Application.Contents(x) & "<br />")
next
%>
```

Output:

test1=First test
test3=Third test

Example 2

```
<%
Application("test1")=("First test")
Application("test2")=("Second test")
Application("test3")=("Third test")

Application.Contents.Remove(2)
```

```
for each x in Application.Contents
  Response.Write(x & "=" & Application.Contents(x) & "<br />")
next
%>
```

Output:

```
test1=First test
test3=Third test
```

2.**Contents.RemoveAll**

The Contents.RemoveAll method deletes all items from the Contents collection.

Syntax

```
Application.Contents.RemoveAll()
```

```
<%
Application.Contents.RemoveAll()
%>
```

3. **Lock and UnLock**

Lock Method

The Lock method prevents other users from modifying the variables in the Application object (used to ensure that only one client at a time can modify the Application variables).

Unlock Method

The Unlock method enables other users to modify the variables stored in the Application object (after it has been locked using the Lock method).

Syntax

```
Application.Lock

Application.Unlock
```

Example

The example below uses the Lock method to prevent more than one user from accessing the variable visits at a time, and the Unlock method to unlock the locked object so that the next client can increment the variable visits:

```
<%
Application.Lock
Application("visits")=Application("visits")+1
Application.Unlock
%>
```

This page has been visited
<%=Application("visits")%> times!

Events
Application_OnEnd

Application_OnStart

Application_OnStart Event

The Application_OnStart event occurs before the first new session is created (when the Application object is first referenced).

This event is placed in the Global.asa file.

Note: Referencing to a Session, Request, or Response objects in the Application_OnStart event script will cause an error.

Application_OnEnd Event

The Application_OnEnd event occurs when the application ends (when the web server stops).

This event is placed in the Global.asa file.

Note: The MapPath method cannot be used in the Application_OnEnd code.

Syntax

```
<script language="vbscript" runat="server">

Sub Application_OnStart
. . .
End Sub

Sub Application_OnEnd
. . .
End Sub

</script>
```

Examples

Global.asa:

```
<script language="vbscript" runat="server">

Sub Application_OnEnd()
Application("totvisitors")=Application("visitors")
```

```
End Sub

Sub Application_OnStart
Application("visitors")=0
End Sub

Sub Session_OnStart
Application.Lock
Application("visitors")=Application("visitors")+1
Application.UnLock
End Sub

Sub Session_OnEnd
Application.Lock
Application("visitors")=Application("visitors")-1
Application.UnLock
End Sub

</script>
```

To display the number of current visitors in an ASP file:

```
<html>
<head>
</head>
<body>
<p>
There are <%response.write(Application("visitors"))%>
online now!
</p>
</body>
</html>
```

The Global.asa file

The Global.asa file is an optional file that can contain declarations of objects, variables, and methods that can be accessed by every page in an ASP application.

All valid browser scripts (JavaScript, VBScript, JScript, PerlScript, etc.) can be used within Global.asa.

The Global.asa file can contain only the following:

- Application events
- Session events
- <object> declarations
- TypeLibrary declarations
- the #include directive

Note: The Global.asa file must be stored in the root directory of the ASP application, and each application can only have one Global.asa file.

Session Object

When you are working with an application on your computer, you open it, do some changes and then you close it. This is much like a Session. The computer knows who you are. It knows when you open the application and when you close it. However, on the internet there is one problem: the web server does not know who you are and what you do, because the HTTP address doesn't maintain state.

ASP solves this problem by creating a unique cookie for each user. The cookie is sent to the user's computer and it contains information that identifies the user. This interface is called the Session object.

The Session object stores information about, or change settings for a user session.

Variables stored in a Session object hold information about one single user, and are available to all pages in one application. Common information stored in session variables are name, id, and preferences. The server creates a new Session object for each new user, and destroys the Session object when the session expires.

The Session object's collections, properties, methods, and events are described below:

Collections

1.Contents

The Contents collection contains all the items appended to the session through a script command.

Tip: To remove items from the Contents collection, use the Remove and RemoveAll methods.

Syntax
Session.Contents(Key)

Example 1

Notice that both name and objtest would be appended to the Contents collection:

```
<%
Session("name")="Hege"
Set Session("objtest")=Server.CreateObject("ADODB.Connection")
%>
```

Example 2

To loop through the Contents collection:

```
<%
for each x in Session.Contents
  Response.Write(x & "=" & Session.Contents(x) & "<br />")
```

```
next
%>
```

or:

```
<%
For i=1 to Session.Contents.Count
  Response.Write(i & "=" & Session.Contents(i) & "<br />")
Next
%>
```

Example 3

```
<%
Session("name")="Hege"
Session("date")="2001/05/05"

for each x in Session.Contents
  Response.Write(x & "=" & Session.Contents(x) & "<br />")
next
%>
```

Output:

```
name=Hege
date=2001/05/05
```

2. StaticObjects

The StaticObjects collection contains all the objects appended to the session with the HTML <object> tag.

Syntax

```
Session.StaticObjects(Key)
```

Parameter	Description
Key	Required. The name of the item to retrieve

Example 1

To loop through the StaticObjects collection:

```
<%
for each x in Session.StaticObjects
  Response.Write(x & "<br />")
next
%>
```

Example 2

In Global.asa:

```
<object runat="server" scope="session"
id="MsgBoard" progid="msgboard.MsgBoard">
</object>

<object runat="server" scope="session"
id="AdRot" progid="MSWC.AdRotator">
</object>
```

In an ASP file:

```
<%
for each x in Session.StaticObjects
  Response.Write(x & "<br />")
next
%>
```

Output:

MsgBoard
AdRot

Properties

1.CodePage

The CodePage property specifies the character set that will be used when displaying dynamic content.

Example of some code pages:

- 1252 - American English and most European languages
- 932 - Japanese Kanji

Syntax
Session.CodePage(=Codepage)

Parameter	Description
Codepage	Defines a code page (character set) for the system running the script engine

Examples

```
<%
Response.Write(Session.CodePage)
%>
```

Output:
1252

2. LCID

The LCID property sets or returns an integer that specifies a location or region. Contents like date, time, and currency will be displayed according to that location or region.

Syntax

Session.LCID(=LCID)

Parameter	Description
LCID	A locale identifier

Examples

```
<%
response.write("<p>")
response.write("Default LCID is: " & Session.LCID & "<br />")
response.write("Date format is: " & date() & "<br />")
response.write("Currency format is: " & FormatCurrency(350))
response.write("</p>")

Session.LCID=1036

response.write("<p>")
response.write("LCID is now: " & Session.LCID & "<br />")
response.write("Date format is: " & date() & "<br />")
response.write("Currency format is: " & FormatCurrency(350))
response.write("</p>")

Session.LCID=3079

response.write("<p>")
response.write("LCID is now: " & Session.LCID & "<br />")
response.write("Date format is: " & date() & "<br />")
response.write("Currency format is: " & FormatCurrency(350))
response.write("</p>")

Session.LCID=2057

response.write("<p>")
response.write("LCID is now: " & Session.LCID & "<br />")
response.write("Date format is: " & date() & "<br />")
response.write("Currency format is: " & FormatCurrency(350))
response.write("</p>")
%>
```

Output:

Default LCID is: 2048

Date format is: 12/11/2001
Currency format is: $350.00

LCID is now: 1036
Date format is: 11/12/2001
Currency format is: 350,00 F

LCID is now: 3079
Date format is: 11.12.2001
Currency format is: öS 350,00

LCID is now: 2057
Date format is: 11/12/2001
Currency format is: £350.00

3.SessionID

The SessionID property returns a unique id for each user. The unique id is generated by the server.

Syntax
Session.SessionID

Examples
<%
Response.Write(Session.SessionID)
%>

Output:

772766038

4. TimeOut

The Timeout property sets or returns the timeout period for the Session object for this application, in minutes. If the user does not refresh or request a page within the timeout period, the session will end.

Syntax
Session.Timeout[=nMinutes]

Parameter	Description
nMinutes	The number of minutes a session can remain idle before the server terminates it. Default is 20 minutes

Examples
<%
response.write("<p>")
response.write("Default Timeout is: " & Session.Timeout)

```
response.write("</p>")

Session.Timeout=30

response.write("<p>")
response.write("Timeout is now: " & Session.Timeout)
response.write("</p>")
%>
```

Output:

Default Timeout is: 20

Timeout is now: 30

Methods

1.Abandon

The Abandon method destroys a user session.

Note: When this method is called, the current Session object is not deleted until all of the script on the current page have been processed. This means that it is possible to access session variables on the same page as the call to Abandon, but not from another Web page.

Syntax
Session.Abandon

Examples

File1.asp:

```
<%
Session("name")="Hege"
Session.Abandon
Response.Write(Session("name"))
%>
```

Output:

Hege

File2.asp:

```
<%
Response.Write(Session("name"))
%>
```

Output:

(none)

2. Contents.Remove

The Contents.Remove method deletes an item from the Contents collection.

Syntax
Session.Contents.Remove(name|index)

Parameter	Description
Name	The name of the item to remove
Index	The index of the item to remove

Example 1

```
<%
Session("test1")=("First test")
Session("test2")=("Second test")
Session("test3")=("Third test")

Session.Contents.Remove("test2")

for each x in Session.Contents
  Response.Write(x & "=" & Session.Contents(x) & "<br />")
next
%>
```

Output:

test1=First test
test3=Third test

Example 2

```
<%
Session("test1")=("First test")
Session("test2")=("Second test")
Session("test3")=("Third test")

Session.Contents.Remove(2)

for each x in Session.Contents
  Response.Write(x & "=" & Session.Contents(x) & "<br />")
next
%>
```

Output:

test1=First test
test3=Third test

3.Contents.RemoveAll

The Contents.RemoveAll method deletes all items from the Contents collection.

Syntax
Session.Contents.RemoveAll()

Example:
<%
Session.Contents.RemoveAll()
%>

Events

Session_OnStart Event

The Session_OnStart event occurs when the server creates a session.

This event is placed in the Global.asa file.

Session_OnEnd Event

The Session_OnEnd event occurs when the session ends (abandoned or times out).

This event is placed in the Global.asa file.

(see Application Object events for the example)

VBSCRIPT

- ✓ Scripting Language
- ✓ Light version of Microsoft's Programming Language Visual Basic
- ✓ Server side and client side both
- ✓ Client side is only supported by Internet Explorer

Client-Side Example

```
<html>

<body>

<script type="text/vbscript">

Document.write("Hello World!")

</script>

</body>

</html>
```

Variables

Declaring variables in ASP is simple, especially since all variables are of Variant type. What does this mean to you? You don't have to declare if your variable is an integer, string, or object. You just declare it, and it has the potential to be anything. To declare a variable in ASP/VBScript we use the Dim statement.

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
'Commented lines starting with an apostrophe
'are not executed in VBScript
'First we will declare a few variables.

Dim myText, myNum
myText = "Have a nice day!"
myNum = 5
Response.Write(myText)

'To concatenate strings in VBScript, use the ampersand
Response.Write(" My favourite number is " & myNum)
%>
```

In ASP/VBScript, it is possible not to declare variables at all. A variable can appear in the program, though it has never been declared. It is called default declaring. Variable in this case will be of Variant type.

However, such practice leads to errors and should be avoided. For VB to consider any form or module variable that was not declared explicitly as erroneous, Option Explicit statement should appear in the form or module main section before any other statements. Option Explicit demands explicit declaration of all variables in this form or module. If module contains Option Explicit statement, than upon the attempt to use undeclared or incorrectly typed variable name, an error occurs at compile time.

In naming variables in VBScript you must be aware of these rules:

- Variables must begin with a letter not a number or an underscore
- They cannot have more than 255 characters
- They cannot contain a period (.) , a space or a dash
- They cannot be a predefined identifier (such as dim, variable, if, etc.)
- Case sensitivity is not important in VBScript

The variable value will be kept in memory for the life span of the current page and will be released from memory when the page has finished executing. To declare variables accessible to more than one ASP file, declare them as session variables or application variables.

Contstants

Constants just as variables are used to store information. The main difference between constants and variables is that constant value can not be changed in the process of running program. If we attempt to re-assign the value of the constant we'll get a run time error.

It can be mathematic constants, passwords, paths to files, etc. By using a constant you "lock in" the value which prevents you from accidentally changing it. If you want to run a program several times using a different value each time, you do not need to search throughout the entire program and change the value at each instance. You only need to change it at the beginning of the program where you set the initial value for the constant.

To declare a constant in VBScript we use the Const keyword. Have a look at the following example:

```
Const myConst = "myText"

'it is allowed to declare a few constants on the one line
Const PI = 3.14159, Wg  = 2.78
```

Conditional Statements

If ... Then ... Else Statement

The If Statement is a way to make decisions based on a variable or some other type of data. For example, you might have a script that checks if Boolean value is true or false or if variable contains number or string value.

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false. The syntax for If statement looks as follows:

```
if condition then
  statements_1
else
```

```
  statements_2
end if
```

Condition can be any expression that evaluates to true or false. If condition evaluates to true, statements_1 are executed; otherwise, statements_2 are executed. statement_1 and statement_2 can be any statement, including further nested if statements.

You may also compound the statements using elseif to have multiple conditions tested in sequence. You should use this construction if you want to select one of many sets of lines to execute.

```
if condition_1 then
  statement_1
[elseif condition_2 then
  statement_2]
...
[elseif condition_n_1 then
  statement_n_1]
[else
  statement_n]
end if
```

Let's have a look at the examples. The first example decides whether a student has passed an exam with a pass mark of 57

```
<%@ language="vbscript"%>
<%
Dim Result
Result = 70

if Result >= 57 then
   response.write("Pass <br />")
else
   response.write("Fail <br />")
end if
%>
```

Next example use the elseif variant on the if statement. This allows us to test for other conditions if the first one wasn't true. The program will test each condition in sequence until:

- It finds one that is true. In this case it executes the code for that condition.
- It reaches an else statement. In which case it executes the code in the else statement.
- It reaches the end of the if ... elseif ... else structure. In this case it moves to the next statement after the conditional structure.

```
<%@ language="vbscript"%>
<%
Dim Result
Result = 70

if Result >= 75 then
   response.write("Passed: Grade A <br />")
elseif Result >= 60 then
```

```
   response.write("Passed: Grade B <br />")
elseif Result >= 45 then
   response.write("Passed: Grade C <br />")
else
   response.write("Failed <br />")
end if
%>
```

Select Case Statement

The Select statements work the same as if statements. However the difference is that they can check for multiple values. Of course you do the same with multiple if..else statements, but this is not always the best approach.

The Select statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement. The syntax for the Select statement as follows:

```
select case expression
  case label_1
    statements_1
  case label_2
    statements_2
  ...
  case else
   statements_n
end select
```

The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements. If no matching label is found, the program looks for the optional Case Else clause, and if found, transfers control to that clause, executing the associated statements. If no Case Else clause is found, the program continues execution at the statement following the end of Select. Use break to prevent the code from running into the next case automatically.

Let's consider an example:

```
<%@ language="vbscript">
<%
Dim Flower
Flower = "rose"

select case flower
 case "rose"
   response.write(flower & " costs $2.50")
 case "daisy"
   response.write(flower & " costs $1.25")
 case "orchild"
   response.write(flower & " costs $1.50")
 case else
   response.write("There is no such flower in our shop")
end select
%>
```

Looping Statements

ASP performs several types of repetitive operations, called "looping". Loops are set of instructions used to repeat the same block of code till a specified condition returns false or true depending on how you need it. To control the loops you can use counter variable that increments or decrements with each repetition of the loop.

The two major groups of loops are For..Next and Do..Loop. The For...Next statements are best used when you want to perform a loop a specific number of times. The Do...Loop statements are best used to perform a loop an undetermined number of times. In addition, you can use the Exit keyword within loop statements.

- The For ... Next Loop
- The For Each ... Next Loop
- The Do ... Loop
- The Exit Keyword

The For ... Next Loop

For...Next loops are used when you want to execute a piece of code a set number of times. The syntax is as follows:

```
For counter = initial_value to  finite_value [Step increment]
  statements
Next
```

The For statement specifies the counter variable and its initial and finite values. The Next statement increases the counter variable by one. Optional the Step keyword allows to increase or decrease the counter variable by the value you specify.

Have a look at the very simple example:

```
<%
For i = 0 to 10 Step 2 'use i as a counter
 response.write("The number is " & i & "<br />")
Next
%>
```

The preceding example prints out even numbers from 0 to 10, the
 tag puts a line break in between each value.

Next example generates a multiplication table 2 through 9. Outer loop is responsible for generating a list of dividends, and inner loop will be responsible for generating lists of dividers for each individual number:

```
<%
response.write("<h1>Multiplication table</h1>")
response.write("<table border=2 width=50%")

For i = 1 to 9            'this is the outer loop
  response.write("<tr>")
  response.write("<td>" & i & "</td>")
```

```
  For j = 2 to 9          'inner loop
     response.write("<td>" & i * j & "</td>")
  Next 'repeat the code and move on to the next value of j

  response.write("</tr>")
Next 'repeat the code and move on to the next value of i

response.write("</table>")
%>
```

The For Each ... Next Loop

The For Each...Next loop is similar to a For...Next loop. Instead of repeating the statements a specified number of times, the For Each...Next loop repeats the statements for each element of an array (or each item in a collection of objects).

The following code snippet creates drop-down list where options are elements of an array:

```
<%
Dim bookTypes(7) 'creates first array
bookTypes(0)="Classic"
bookTypes(1)="Information Books"
bookTypes(2)="Fantasy"
bookTypes(3)="Mystery"
bookTypes(4)="Poetry"
bookTypes(5)="Humor"
bookTypes(6)="Biography"
bookTypes(7)="Fiction"

Dim arrCars(4) 'creates second array
arrCars(0)="BMW"
arrCars(1)="Mercedes"
arrCars(2)="Audi"
arrCars(3)="Bentley"
arrCars(4)="Mini"

Sub createList(some_array) 'takes an array and creates drop-down list
  dim i
  response.write("<select name=""mylist"">" & vbCrLf) 'vbCrLf stands for Carriage Return and Line
Feed
  For Each item in some_array
     response.write("<option value=" & i & ">" & item & "</option>" & vbCrLf)
     i = i + 1
  Next 'repeat the code and move on to the next value of i
  response.write("</select>")
End Sub

'Now let's call the sub and print out our lists on the screen
Call createList(bookTypes) 'takes bookTypes array as an argument
Call createList(arrcars) 'takes arrCars array as an argument
%>
```

The Do..while Loop

The Do...Loop is another commonly used loop after the For...Next loop. The Do...Loop statement repeats a block of statements an indefinite number of times. The statements are repeated either while a condition is True or until a condition becomes True. The syntax looks as follows:

```
Do [While|Until] condition
  statements
Loop
```

Here is another syntax:

```
Do
  statements
Loop [While|Until] condition
```

In this case the code inside this loop will be executed at least one time. Have a look at the examples:

The example below defines a loop that starts with i=0. The loop will continue to run as long as i is less than, or equal to 10. i will increase by 1 each time the loop runs.

```
<%
Dim i 'use i as a counter
i = 0 'assign a value to i

Do While i<=10 'Output the values from 0 to 10
  response.write(i & "<br \>")
  i = i + 1 'increment the value of i for next time loop executes
Loop
%>
```

Now let's consider a more useful example which creates drop-down lists of days, months and years. You can use this code for registration form, for example.

```
<%
'creates an array
Dim month_array(11)
month_array(0) = "January"
month_array(1) = "February"
month_array(2) = "March"
month_array(3) = "April"
month_array(4) = "May"
month_array(5) = "June"
month_array(6) = "July"
month_array(7) = "August"
month_array(8) = "September"
month_array(9) = "October"
month_array(10) = "November"
month_array(11) = "December"

Dim i
response.write("<select name=""day"">" & vbCrLf)
i = 1
```

```
Do While i <= 31
  response.write("<option value=" & i & ">" & i & "</option>" & vbCrLf)
  i = i + 1
Loop
response.write("</select>")

response.write("<select name=""month"">" & vbCrLf)
i = 0
Do While i <= 11
  response.write("<option value=" & i & ">" & month_array(i) & "</option>" & vbCrLf)
  i = i + 1
Loop
response.write("</select>")

response.write("<select name=""year"">")
i = 1900
Do Until i = 2005
  response.write("<option value=" & i & ">" & i & "</option>" & vbCrLf)
  i = i + 1
Loop
response.write("</select>")
%>
```

Note: Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate.

The Exit Keyword

The Exit keyword alters the flow of control by causing immediate exit from a repetition structure. You can use the Exit keyword in different situations, for example to avoid an endless loop. To exit the For...Next loop before the counter reaches its finite value you should use the Exit For statement. To exit the Do...Loop use the Exit Do statement.

Have a look at the example:

```
<%
response.write("<p><strong>Example of using the Exit For statement:</strong><p>")

For i = 0 to 10
  If i=3 Then Exit For
  response.write("The number is " & i & "<br />")
Next

response.write("<p><strong>Example of using the Exit Do statement:</strong><p>")

i = 5
Do Until i = 10
 i = i - 1
 response.write("The number is " & i & "<br />")
 If i < 10 Then Exit Do
Loop
%>
```

Arrays

The VBScript arrays are 0 based, meaning that the array element indexing starts always from 0. The 0 index represents the first position in the array, the 1 index represents the second position in the array, and so forth.

There are two types of VBScript arrays - static and dynamic. Static arrays remain with fixed size throughout their life span. To use static VBScript arrays you need to know upfront the maximum number of elements this array will contain. If you need more flexible VBScript arrays with variable index size, then you can use dynamic VBScript arrays. VBScript dynamic arrays index size can be increased/decreased during their life span.

Static Arrays

Let's create an array called 'arrCars' that will hold the names of 5 cars:

```
<%
'Use the Dim statement along with the array name
'to create a static VBScript array
'The number in parentheses defines the array's upper bound
Dim arrCars(4)
arrCars(0)="BMW"
arrCars(1)="Mercedes"
arrCars(2)="Audi"
arrCars(3)="Bentley"
arrCars(4)="Mini"

'create a loop moving through the array
'and print out the values
For i=0 to 4
response.write arrCars(i) & "<br>"
Next     'move on to the next value of i
%>
```

Here is another way to define the array in VBScript:

```
<%
'we use the VBScript Array function along with a Dim statement
'to create and populate our array
Dim arrCars
arrCars = Array("BMW","Mercedes","Audi","Bentley","Mini") 'each element must be separated by a comma

'again we could loop through the array and print out the values
For i=0 to 4
response.write arrCars(i) & "<br>"
Next
%>
```

Dynamic Arrays

Dynamic arrays come in handy when you aren't sure how many items your array will hold. To create a dynamic array you should use the Dim statement along with the array's name, without specifying upper bound:

```
<%
Dim arrCars
arrCars = Array()
%>
```

In order to use this array you need to use the ReDim statement to define the array's upper bound:

```
<%
Dim arrCars
arrCars = Array()
Redim arrCars(27)
%>
```

If in future you need to resize this array you should use the Redim statement again. Be very careful with the ReDim statement. When you use the ReDim statement you lose all elements of the array. Using the keyword PRESERVE in conjunction with the ReDim statement will keep the array we already have and increase the size:

```
<%
Dim arrCars
arrCars = Array()
Redim arrCars(27)
Redim PRESERVE arrCars(52)
%>
```

Multidimensional Arrays

Arrays do not have to be a simple list of keys and values; each location in the array can hold another array. This way, you can create a multi-dimensional array.

The most commonly used are two-dimensional arrays. You can think of a two-dimensional array as a matrix, or grid, with width and height or rows and columns. Here is how you could define two-dimensional array and display the array values on the web page:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Dim arrCars(2,4)

'arrCars(col,row)
arrCars(0,0) = "BMW"
arrCars(1,0) = "2004"
arrCars(2,0) = "45.000"
arrCars(0,1) = "Mercedes"
arrCars(1,1) = "2003"
arrCars(2,1) = "57.000"
arrCars(0,2) = "Audi"
arrCars(1,2) = "2000"
arrCars(2,2) = "26.000"
```

```
arrCars(0,3) = "Bentley"
arrCars(1,3) = "2005"
arrCars(2,3) = "100.00"
arrCars(0,4) = "Mini"
arrCars(1,4) = "2004"
arrCars(2,4) = "19.00"

Response.Write(" <TABLE border=0>")
Response.Write("<TR><TD>Row</TD> <TD>Car</TD>")
Response.Write("<TD>Year</TD><TD>Price</TD></TR>")

'The UBound function will return the 'index' of the highest element in an array.
For i = 0 to UBound(arrCars, 2)
Response.Write("<TR><TD>#" & i & "</TD>")
Response.Write("<TD>" & arrCars(0,i) & "</TD>")
Response.Write("<TD>" & arrCars(1,i) & "</TD>")
Response.Write("<TD>" & arrCars(2,i) & "</TD></TR>")
Next

Response.Write("</TABLE>")

%>
```

OR You can loop the above array as:

```
For i=0 to ubound(arrCars,1)

 For j=0 to ubound(arrCars,2)

        Response.write(arrCars(i,j) & "<br/>")

  Next

Next
```

Functions and Procedures

Functions and procedures provide a way to create re-usable modules of programming code and avoid rewriting the same block of code every time you do the particular task. If you don't have any functions/procedures in your ASP page, the ASP pages are executed from top to bottom, the ASP parsing engine simply processes your entire file from the beginning to the end. VBScript functions and procedures, however, are executed only when called, not inline with the rest of the code. A function or procedure can be reused as many times as required, thus saving you time and making for a less clustered looking page.

You can write functions in ASP similar to the way you write them in Visual Basic. It is good programming practice to use functions to modularize your code and to better provide reuse. To declare a subroutine (a function that doesn't return a value, starts with the Sub keyword and ends with End Sub), you simply type:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
```

```
Sub subroutineName( parameter_1, ... , parameter_n )
  statement_1
  statement_2
   ...
  statement_n
end sub
%>
```

A function differs from a subroutine in the fact that it returns data, start with Function keyword and end with End Function. Functions are especially good for doing calculations and returning a value. To declare a function, the syntax is similar:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Function  functionName( parameter_1, ... , parameter_n )
  statement_1
  statement_2
   ...
  statement_n
end function
%>
```

Have a look at the code for a procedure that is used to print out information on the page:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Sub GetInfo(name, phone, fee)
  Response.write("Name: "& name &"<br>")
  Response.write("Telephone: "& telephone &"<br>")
  Response.write("Fee: "& fee &"<br>")
End Sub
%>
```

Now let's consider how to call the sub. There are two ways:

```
<%
'the first method
Call GetInfo("Mr. O'Donnel","555-5555",20)
'the second one
GetInfo "Mr. O'Donnel","555-5555",20
%>
```

In each example, the actual argument passed into the subprocedure is passed in the corresponding position. Note that if you use the Call statement, the arguments must be enclosed in parentheses. If you do not use call, the parentheses aren't used.

Now let's look at the code for a function that takes an integer value and returns the square of that value. Also included is code to call the function.

```
<%
Function Square(num)
  Square = num * num
end function
```

```
 'Returns 25
 Response.Write(Square(5))

 'Should print "45 is less than 8^2"
 if 40 < Square(7) then
  Response.Write("45 is less than 8^2")
 else
  Response.Write("8^2 is less than 40")
 end if
%>
```

How to process the data submitted from HTML form

The great advantage of ASP is possibility to respond to user queries or data submitted from HTML forms. You can process information gathered by an HTML form and use ASP code to make decisions based off this information to create dynamic web pages. In this tutorial we will show how to create an HTML form and process the data.

Before you can process the information, you need to create an HTML form that will send information to your ASP page. There are two methods for sending data to an ASP form: POST and GET. These two types of sending information are defined in your HTML form element's method attribute. Also, you must specify the location of the ASP page that will process the information.

Below is a simple form that will send the data using the POST method. Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send. Copy and paste this code and save it as "form.html".

```
<html>
<head>
<title>Process the HTML form data with the POST method</title>
</head>
<body>
<form method="POST" action="process.asp" name="form1">
<table width="70%" border="0" cellspacing="0" cellpadding="0">
<tr>
  <td>name:</td>
  <td colspan="2"><input type="text" name="name"></td>
</tr>
<tr>
  <td>email:</td>
  <td colspan="2"><input type="text" name="email"></td>
</tr>
<tr>
  <td>comments:</td>
  <td colspan="2"><textarea name="comment" cols="40" rows="5"></textarea></td>
</tr>
<tr>
  <td> </td>
  <td colspan="2"><input type="submit" name="Submit" value="Submit"></td>
</tr>
</table>
```

```
</form>
</body>
</html>
```

Next, we are going to create our ASP page "process.asp" that will process the data. In our example we decided to send data with the POST method so to retrieve the information we can use the ASP 'Request.From' command. Copy and paste this code and save it in the same directory as "form.html".

```
<%@ Language="VBscript" %>
<html>
<head>
<title>Submitted data</title>
</head>

<body>
<%
'declare the variables that will receive the values
Dim name, email, comment
'receive the values sent from the form and assign them to variables
'note that request.form("name") will receive the value entered
'into the textfield called name
name=Request.Form("name")
email=Request.Form("email")
comment=Request.Form("comment")

'let's now print out the received values in the browser
Response.Write("Name: " & name & "<br>")
Response.Write("E-mail: " & email & "<br>")
Response.Write("Comments: " & comment & "<br>")
%>
</body>
</html>
```

Note: If you want to process the information sent through an HTML form with the GET method you should use the 'Request.QueryString' command . In the preceding example you should replace all instances of Form with QueryString. But remember that the data sent from a form with the GET method is visible to everyone (it will be displayed in the browser's address bar) and has limits on the amount of information to send.

The Dictionary Object

The Dictionary object is used to store information in name/value pairs (referred to as key and item). The Dictionary object might seem similar to Arrays, however, the Dictionary object is a more desirable solution to manipulate related data.

Comparing Dictionaries and Arrays:

- Keys are used to identify the items in a Dictionary object
- You do not have to call ReDim to change the size of the Dictionary object

- When deleting an item from a Dictionary, the remaining items will automatically shift up
- Dictionaries cannot be multidimensional, Arrays can
- Dictionaries have more built-in functions than Arrays
- Dictionaries work better than arrays on accessing random elements frequently
- Dictionaries work better than arrays on locating items by their content

The following example creates a Dictionary object, adds some key/item pairs to it, and retrieves the item value for the key gr:

```
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
Response.Write("The value of key gr is: " & d.Item("gr"))
%>
```

Output:

The value of key gr is: Green

The Dictionary object's properties and methods are described below:

Properties

1.CompareMode

The CompareMode property sets or returns the comparison mode for comparing keys in a Dictionary object.

Syntax
DictionaryObject.CompareMode[=compare]

Parameter	Description
Compare	Optional. Specifies the comparison mode. Can take one of the following values: 0 = vbBinaryCompare - binary comparison 1 = vbTextCompare - textual comparison 2 = vbDatabaseCompare - database comparison

Example

```
<%
dim d
set d=Server.CreateObject("Scripting.Dictionary")
d.CompareMode=1
d.Add "n","Norway"
d.Add "i","Italy"

'The Add method will fail on the line below!

d.Add "I","Ireland" 'The letter i already exists
%>
```

2.Count

The Count property returns the number of key/item pairs in the Dictionary object.

Syntax

```
DictionaryObject.Count
```

Example

```
<%
dim d
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"
Response.Write("The number of key/item pairs: " & d.Count)
set d=nothing
%>
```

Output:

The number of key/item pairs: 3

3. Item

The Item property sets or returns the value of an item in a Dictionary object.

Syntax

```
DictionaryObject.Item(key)[=newitem]
```

Parameter	Description

Key	Required. The key associated with the item
Newitem	Optional. Specifies the value associated with the key

Example
```
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
Response.Write("The value of key pi is: " & d.Item("pi"))
%>
```

Output:

The value of key pi is: Pink

4. Key

The Key property sets a new key value for an existing key value in a Dictionary object.

Syntax
```
DictionaryObject.Key(key)=newkey
```

Parameter	**Description**
Key	Required. The name of the key that will be changed
Newkey	Required. The new name of the key

Example
```
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
d.Key("re")="r"
Response.Write("The value of key r is: " & d.Item("r"))
```

%>

Output:

The value of key r is: Red

Methods

1.Add

The Add method adds a new key/item pair to a Dictionary object.

Syntax
DictionaryObject.Add(key,item)

Parameter	Description
key	Required. The key value associated with the item
item	Required. The item value associated with the key

Example
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
Response.Write("The value of key gr is: " & d.Item("gr"))
%>
Output:
The value of key gr is: Green

2. Exists

The Exists method returns a Boolean value that indicates whether a specified key exists in the Dictionary object. It returns true if the key exists, and false if not.

Syntax
DictionaryObject.Exists(key)

Parameter	Description

key	Required. The key value to search for

Example
```
<%
dim d
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

if d.Exists("n")=true then
  Response.Write("Key exists!")
else
  Response.Write("Key does not exist!")
end if

set d=nothing
%>
```

Output:

Key exists!

3.Items

The Items method returns an array of all the items in a Dictionary object.

Syntax
DictionaryObject.Items

Example
```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

Response.Write("<p>Item values:</p>")
a=d.Items
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next
```

```
set d=nothing
%>
```
Output:
Item values:
Norway
Italy
Sweden

4.Keys

The Keys method returns an array of all the keys in a Dictionary object.

Syntax
DictionaryObject.Keys

Example
```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

Response.Write("<p>Key values:</p>")
a=d.Keys
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next

set d=nothing
%>
```

Output:

Key values:
n
i
s

5.Remove

The Remove method removes one specified key/item pair from the Dictionary object.

Syntax
DictionaryObject.Remove(key)

Parameter	Description
key	Required. The key associated with the key/item pair to remove

Example

```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

d.Remove("n")

Response.Write("<p>Key values:</p>")
a=d.Keys
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next

set d=nothing
%>
```

Output:

```
Key values:
i
s
```

6.RemoveAll

The Remove method removes all the key/item pairs from a Dictionary object.

Syntax

```
DictionaryObject.RemoveAll
```

Example

```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

d.RemoveAll
```

```
Response.Write("<p>Key values:</p>")
a=d.Keys
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next
set d=nothing
%>
```

Output:

Key values:

(nothing)

FileSystemObject Object

The FileSystemObject object is used to access the file system on a server.This object can manipulate files, folders, and directory paths. It is also possible to retrieve file system information with this object.The following code creates a text file (c:\test.txt) and then writes some text to the file:

```
<%
dim fs,fname
set fs=Server.CreateObject("Scripting.FileSystemObject")
set fname=fs.CreateTextFile("c:\test.txt",true)
fname.WriteLine("Hello World!")
fname.Close
set fname=nothing
set fs=nothing
%>
```

The FileSystemObject object's properties and methods are described below:

Properties

Drives

The Drives property returns a collection of all Drive objects on the computer.

Methods

1.BuildPath

The BuildPath method appends a name to an existing path.

Syntax

[newpath=]FileSystemObject.BuildPath(path,name)

Parameter	Description
Path	Required. The path to append a name to
Name	Required. The name to append to the path

Example
```
<%
dim fs,path
set fs=Server.CreateObject("Scripting.FileSystemObject")
path=fs.BuildPath("c:\mydocuments","test")
response.write(path)
set fs=nothing
%>
```
Output:
c:\mydocuments\test

2.CopyFile

The CopyFile method copies one or more files from one location to another.

Syntax
FileSystemObject.CopyFile source,destination[,overwrite]

Parameter	Description
Source	Required. The file or files to copy (wildcards can be used}
destination	Required. Where to copy the file or files (wildcards cannot be used}
overwrite	Optional. A Boolean value that specifies whether an existing file can be overwritten. True allows existing files to be overwritten and False prevents existing files from being overwritten. Default is True

Example
```
<%
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CopyFile "c:\mydocuments\web\*.htm","c:\webpages\"
set fs=nothing
%>
```

3.CopyFolder

The CopyFolder method copies one or more folders from one location to another.

Syntax
FileSystemObject.CopyFolder source,destination[,overwrite]

Parameter	Description
source	Required. The folder or folders to copy (wildcards can be used)
destination	Required. Where to copy the folder or folders (wildcards cannot be used)
overwrite	Optional. A Boolean value that indicates whether an existing folder can be overwritten. True allows existing folders to be overwritten and False prevents existing folders from being overwritten. Default is True

Examples

```
<%
'copy all the folders in c:\mydocuments\web
'to the folder c:\webpages

dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CopyFolder "c:\mydocuments\web\*","c:\webpages\"
set fs=nothing
%>
<%
'copy only the folder test from c:\mydocuments\web
'to the folder c:\webpages
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CopyFolder "c:\mydocuments\web\test","c:\webpages\"
set fs=nothing
%>
```

4.CreateFolder

Creates a new folder

5.CreateTextFile

The CreateTextFile method creates a new text file in the current folder and returns a TextStream object that can be used to read from, or write to the file.

Syntax
FileSystemObject.CreateTextFile(filename[,overwrite[,unicode]])

FolderObject.CreateTextFile(filename[,overwrite[,unicode]])

Parameter	Description
Filename	Required. The name of the file to create
Overwrite	Optional. A Boolean value that indicates whether an existing file can be overwritten. True indicates that the file can be overwritten and False indicates that the file can not be overwritten. Default is True
Unicode	Optional. A Boolean value that indicates whether the file is created as a Unicode or an ASCII file. True indicates that the file is created as a Unicode file, False indicates that the file is created as an ASCII file. Default is False

Example for the FileSystemObject object

```
<%
dim fs,tfile
set fs=Server.CreateObject("Scripting.FileSystemObject")
set tfile=fs.CreateTextFile("c:\somefile.txt")
tfile.WriteLine("Hello World!")
tfile.close
set tfile=nothing
set fs=nothing
%>
```

Example for the Folder object

```
<%
dim fs,fo,tfile
Set fs=Server.CreateObject("Scripting.FileSystemObject")
Set fo=fs.GetFolder("c:\test")
Set tfile=fo.CreateTextFile("test.txt",false)
tfile.WriteLine("Hello World!")
tfile.Close
set tfile=nothing
set fo=nothing
set fs=nothing
%>
```

6.DeleteFile

The DeleteFile method deletes one or more specified files.

Note: An error will occur if you try to delete a file that doesn't exist.

Syntax
FileSystemObject.DeleteFile(filename[,force])

Parameter	Description
filename	Required. The name of the file or files to delete (Wildcards are allowed)
force	Optional. A Boolean value that indicates whether read-only files will be deleted. True indicates that the read-only files will be deleted, False indicates that they will not be deleted. Default is False

Example

```
<%
dim fs
Set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CreateTextFile("c:\test.txt",True)
if fs.FileExists("c:\test.txt") then
  fs.DeleteFile("c:\test.txt")
end if
set fs=nothing
%>
```

7.DeleteFolder

The DeleteFolder method deletes one or more specified folders.

Note: An error will occur if you try to delete a folder that does not exist.

Syntax
FileSystemObject.DeleteFolder(foldername[,force])

Parameter	Description
foldername	Required. The name of the folder or folders to delete (Wildcards are allowed)
force	Optional. A Boolean value that indicates whether read-only folders will be deleted. True indicates that read-only folders will be deleted, False indicates that they will not be deleted. Default is False

Example

```
<%
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
if fs.FolderExists("c:\temp") then
  fs.DeleteFolder("c:\temp")
end if
set fs=nothing
%>
```

8.GetFile

The GetFile method returns a File object for the specified path.

Syntax
FileSystemObject.GetFile(path)

Parameter	Description
path	Required. The path to a specific file

Example

```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.GetFile("c:\test\test.htm")
Response.Write("The file was last modified on: ")
Response.Write(f.DateLastModified)
set f=nothing
set fs=nothing
%>
```

Output:
The file was last modified on 01/01/20 4:23:56 AM

9.**MoveFile**

The MoveFile method moves one or more files from one location to another.

Syntax
FileSystemObject.MoveFile source,destination

Parameter	Description
source	Required. The path to the file/files to be moved. Can contain wildcard characters in

	the last component.
destination	Required. Where to move the file/files. Cannot contain wildcard characters

Example
<%
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.MoveFile "c:\web*.gif","c:\images\"
set fs=nothing
%>

10.**OpenTextFile**

The OpenTextFile method opens a specified file and returns a TextStream object that can be used to access the file.

Syntax
FileSystemObject.OpenTextFile(fname,mode,create,format)

Parameter	**Description**
fname	Required. The name of the file to open
mode	Optional. How to open the file 1=ForReading - Open a file for reading. You cannot write to this file. 2=ForWriting - Open a file for writing. 8=ForAppending - Open a file and write to the end of the file.
create	Optional. Sets whether a new file can be created if the filename does not exist. True indicates that a new file can be created, and False indicates that a new file will not be created. False is default
format	Optional. The format of the file 0=TristateFalse - Open the file as ASCII. This is default. -1=TristateTrue - Open the file as Unicode. -2=TristateUseDefault - Open the file using the system default.

Example
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.OpenTextFile(Server.MapPath("testread.txt"),8,true)

```
f.WriteLine("This text will be added to the end of file")
f.Close
set f=Nothing
set fs=Nothing
%>
```

The TextStream Object

The TextStream object is used to access the contents of text files.

The following code creates a text file (c:\test.txt) and then writes some text to the file (the variable f is an instance of the TextStream object):

```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt",true)
f.WriteLine("Hello World!")
f.Close
set f=nothing
set fs=nothing
%>
```

To create an instance of the TextStream object you can use the CreateTextFile or OpenTextFile methods of the FileSystemObject object, or you can use the OpenAsTextStream method of the File object.

The TextStream object's properties and methods are described below:

Properties

Property	Description
AtEndOfLine	Returns true if the file pointer is positioned immediately before the end-of-line marker in a TextStream file, and false if not
AtEndOfStream	Returns true if the file pointer is at the end of a TextStream file, and false if not
Column	Returns the column number of the current character position in an input stream
Line	Returns the current line number in a TextStream file

Example:

```
<%
dim fs,f,t,x
set fs=Server.CreateObject("Scripting.FileSystemObject")
```

```
set f=fs.CreateTextFile("c:\test.txt")
f.write("Hello World!")
f.close

set t=fs.OpenTextFile("c:\test.txt",1,false)
do while t.AtEndOfLine<>true
  x=t.Read(1)
loop
t.close
Response.Write("The last character is: " & x)
%>
```

Output:
The last character of the first line in the text file is: !

Methods

1.Close

The Close method closes an open TextStream file.

2. Read

The Read method reads a specified number of characters from a TextStream file and returns the result as a string.

Syntax
TextStreamObject.Read(numchar)

Parameter	Description
Numchar	Required. The number of characters to read from the file

Example

```
<%
dim fs,f,t,x
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt")
f.write("Hello World!")
f.close

set t=fs.OpenTextFile("c:\test.txt",1,false)
x=t.Read(5)
t.close
```

```
Response.Write("The first five characters are: " & x)
%>
```
Output:
The first five characters are: Hello

3.ReadAll

Reads an entire TextStream file and returns the result

4.ReadLine

The ReadLine method reads one line from a TextStream file and returns the result as a string.

Syntax
TextStreamObject.ReadLine

Example
```
<%
dim fs,f,t,x
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt")
f.writeline("Line 1")
f.writeline("Line 2")
f.writeline("Line 3")
f.close
set t=fs.OpenTextFile("c:\test.txt",1,false)
x=t.ReadLine
t.close
Response.Write("The first line in the file ")
Response.Write("contains this text: " & x)
%>
```
Output:
The first line in the file contains this text: Line 1

5.Write

The Write method writes a specified text to a TextStream file.

Note: This method write text to the TextStream file with no spaces or line breaks between each string.

Syntax
TextStreamObject.Write(text)

Parameter	Description

Text	Required. The text to write to the file

Example
```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt",true)
f.write("Hello World!")
f.write("How are you today?")
f.close
set f=nothing
set fs=nothing
%>
```

The file test.txt will look like this after executing the code above:

Hello World!How are you today?

6.WriteLine

The WriteLine method writes a specified text and a new-line character to a TextStream file.

Syntax
TextStreamObject.WriteLine(text)

Parameter	Description
text	Optional. The text to write to the file. If you do not specify this parameter, a new-line character will be written to the file

Example
```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt",true)
f.WriteLine("Hello World!")
f.WriteLine("How are you today?")
f.WriteLine("Goodbye!")
f.close
set f=nothing
set fs=nothing
%>
```

The file test.txt will look like this after executing the code above:

Hello World!
How are you today?
Goodbye!

The File Object

The File object is used to return information about a specified file.

To work with the properties and methods of the File object, you will have to create an instance of the File object through the FileSystemObject object. First; create a FileSystemObject object and then instantiate the File object through the GetFile method of the FileSystemObject object or through the Files property of the Folder object.

The following code uses the GetFile method of the FileSystemObject object to instantiate the File object and the DateCreated property to return the date when the specified file was created:

Example

```
<%
Dim fs,f
Set fs=Server.CreateObject("Scripting.FileSystemObject")
Set f=fs.GetFile("c:\test.txt")
Response.Write("File created: " & f.DateCreated)
set f=nothing
set fs=nothing
%>
```

The File object's properties and methods are described below:

Properties

Property	Description
Attributes	Sets or returns the attributes of a specified file
DateCreated	Returns the date and time when a specified file was created
DateLastAccessed	Returns the date and time when a specified file was last accessed
DateLastModified	Returns the date and time when a specified file was last modified

Drive	Returns the drive letter of the drive where a specified file or folder resides
Name	Sets or returns the name of a specified file
ParentFolder	Returns the folder object for the parent of the specified file
Path	Returns the path for a specified file
ShortName	Returns the short name of a specified file (the 8.3 naming convention)
ShortPath	Returns the short path of a specified file (the 8.3 naming convention)
Size	Returns the size, in bytes, of a specified file
Type	Returns the type of a specified file

Methods

Method	**Description**
Copy	Copies a specified file from one location to another
Delete	Deletes a specified file
Move	Moves a specified file from one location to another
OpenAsTextStream	Opens a specified file and returns a TextStream object to access the file

Debugging ASP and Error Handling

Regardless of your level of experience, you will encounter programmatic errors, or *bugs*, that will prevent your server-side scripts from working correctly. For this reason, debugging, the process of finding and correcting scripting errors, is crucial for developing successful and robust ASP applications, especially as the complexity of your application grows. Various tools like Visual Studio can be used to debug ASP Applications.

The Microsoft Script Debugger

The Microsoft Script Debugger is a powerful debugging tool that can help you quickly locate bugs and interactively test your server-side scripts. You can do the following things using the features of Microsoft Script Debugger:

- Run your server-side scripts one line at a time.

- Open a command window to monitor the value of variables, properties, or array elements, during the execution of your server-side scripts.
- Set pauses to suspend execution of your server-side scripts (using either the debugger or a script command) at a particular line of script.
- Trace procedures while running your server-side script.

Enabling Debugging

Before you can begin debugging your server-side scripts, you must first configure your Web server to support ASP debugging.

After enabling Web server debugging, you can use either of the following methods to debug your scripts:

- Open Script Debugger and use it to run and debug your ASP server-side scripts.
- Use Internet Explorer to request an .asp file. If the file contains a bug or an intentional statement to halt execution, Script Debugger will automatically start, display your script, and indicate the source of the error.

Scripting Errors

While debugging your server-side scripts you might encounter several types of errors. Some of these errors can cause your scripts to execute incorrectly, halt the execution of your program, or return incorrect results.

Syntax Errors

A *syntax* error is a commonly encountered error that results from incorrect scripting syntax. For example, a misspelled command or an incorrect number of arguments passed to a function generates an error. Syntax errors can prevent your script from running.

Runtime Errors

Run-time errors occur after your script commences execution and result from scripting instructions that attempt to perform impossible actions. For example, the following script contains a function that divides a variable by zero (an illegal mathematical operation) and generates a run-time error:

```
<SCRIPT Language= "VBScript" RUNAT=SERVER>
  Result = Findanswer(15)
  Document.Write ("The answer is " &Result)

  Function Findanswer(x)
  'This statement generates a run-time error.
   Findanswer = x/0
  End Function
</SCRIPT>
```

Bugs that result in run-time errors must be corrected for your script to execute without interruption.

Logical Errors

A *logical* error can be the most difficult bug to detect. With logical errors, which are caused by typing mistakes or flaws in programmatic logic, your script runs successfully, but yields incorrect results. For example, a server-side script intended to sort a list of values may return an inaccurate ordering if the script contains a > (greater than) sign for comparing values, when it should have used a < (less than) sign.

Just-In-Time(JIT) Debugging

When a run-time error interrupts execution of your server-side script, the Microsoft Script Debugger automatically starts, displays the .asp file with a statement pointer pointing to the line that caused the error, and generates an error message. With this type of debugging, called? *Just-In-Time* (JIT) debugging, your computer suspends further execution of the program. You must correct the errors with an editing program and save your changes before you can resume running the script.

Breakpoint Debugging

When an error occurs and you cannot easily locate the source of the error, it is sometimes useful to preset a *breakpoint*. A breakpoint suspends execution at a specific line in your script. You can set one or many different breakpoints in Microsoft Script Debugger before a suspect line of script and then use the debugger to inspect the values of variables or properties set in the script. After you correct the error, you can clear your breakpoints so that your script can run uninterrupted.

To set a breakpoint, open your script with Script Debugger, select a line of script where you want to interrupt execution, and from the **Debug** menu choose **Toggle Breakpoint**. Then use your Web browser to request the script again. After executing the lines of script up to the breakpoint, your computer starts the Script Debugger, which displays the script with a statement pointer pointing to the line where you set the breakpoint.

The Break at Next Statement

In certain cases, you may want to enable the Script Debugger **Break at Next Statement** if the next statement that runs is not in the .asp file that you are working with. For example, if you set **Break at Next Statement** in an .asp file residing in an application called Sales, the debugger will start when you run a script in any file in the Sales application, or in any application for which debugging has been enabled. For this reason, when you set **Break at Next Statement**, you need to be aware that whatever script statement runs next will start the debugger.

VBScript Stop Statement Debugging

You can also add breakpoints to your server-side scripts written in VBScript by inserting a **Stop** statement at a location before a questionable section of server-side script. For example, the following server-side script contains a**Stop** statement that suspends execution before the script calls a custom function:

VBScript

```
<%
  intDay = Day(Now())
  lngAccount = Request.Form("AccountNumber")
  dtmExpires = Request.Form("ExpirationDate")
```

```
  strCustomerID  =  "RETAIL" & intDay & lngAccount & dtmExpires

  'Set breakpoint here.
  Stop

  'Call registration component.
  RegisterUser(strCustomerID)
%>
```

When you request this script, the debugger starts and automatically displays the .asp file with the statement pointer indicating the location of the **Stop** statement. At this point you could choose to inspect the values assigned to variables before passing those variables to the component.

The ASPError Object

The ASPError object was implemented in ASP 3.0 and is available in IIS5 and later.

The ASPError object is used to display detailed information of any error that occurs in scripts in an ASP page.

Note: The ASPError object is created when Server.GetLastError is called, so the error information can only be accessed by using the Server.GetLastError method.

The ASPError object's properties are described below (all properties are read-only):

Properties

Property	Description
ASPCode	Returns an error code generated by IIS
ASPDescription	Returns a detailed description of the error (if the error is ASP-related)
Category	Returns the source of the error (was the error generated by ASP? By a scripting language? By an object?)
Column	Returns the column position within the file that generated the error
Description	Returns a short description of the error
File	Returns the name of the ASP file that generated the error
Line	Returns the line number where the error was detected
Number	Returns the standard COM error code for the error

Source	Returns the actual source code of the line where the error occurred

Error Handling

Handling errors is achieved by using the **On Error Resume Next** statement**.** It simply tells the ASP interpreter to continue the execution of the ASP Script if there is an error instead of throwing an exception and stopping the execution.

To trap and handle the errors, we need to use the Err Object and its properties

<% if Err.Number <> 0 Then

Handle_Error(Err.Description)

Err.Clear

End If

%>

<% Sub Handle_Error(errordesc)

'Write error to a log file

%>

On Error GoTo 0 Statement is used to disable any error handling

Error handling allows you to display friendly error messages for the end users and the same time it helps you debug the asp application.

Browser Capabilities Component

The ASP Browser Capabilities component creates a BrowserType object that determines the type, capabilities and version number of a visitor's browser.

When a browser connects to a server, a User Agent header is also sent to the server. This header contains information about the browser.

The BrowserType object compares the information in the header with information in a file on the server called "Browscap.ini".

If there is a match between the browser type and version number in the header and the information in the "Browsercap.ini" file, the BrowserType object can be used to list the properties of the matching browser. If there is no match for the browser type and version number in the Browscap.ini file, it will set every property to "UNKNOWN".

Syntax

```
<%
Set MyBrow=Server.CreateObject("MSWC.BrowserType")
%>
```

ASP Browser Capabilities Example

The example below creates a BrowserType object in an ASP file, and displays some of the capabilities of your browser:

Example

```
<html>
<body>
<%
Set MyBrow=Server.CreateObject("MSWC.BrowserType")
%>

<table border="0" width="100%">
<tr>
<th>Client OS</th><th><%=MyBrow.platform%></th>
</tr><tr>
<td >Web Browser</td><td ><%=MyBrow.browser%></td>
</tr><tr>
<td>Browser version</td><td><%=MyBrow.version%></td>
</tr><tr>
<td>Frame support?</td><td><%=MyBrow.frames%></td>
</tr><tr>
<td>Table support?</td><td><%=MyBrow.tables%></td>
</tr><tr>
<td>Sound support?</td><td><%=MyBrow.backgroundsounds%></td>
</tr><tr>
```

```
<td>Cookies support?</td><td><%=MyBrow.cookies%></td>
</tr><tr>
<td>VBScript support?</td><td><%=MyBrow.vbscript%></td>
</tr><tr>
<td>JavaScript support?</td><td><%=MyBrow.javascript%></td>
</tr>
</table>

</body>
</html>
```

Output:

Client OS	**WinNT**
Web Browser	IE
Browser version	5.0
Frame support?	True
Table support?	True
Sound support?	True
Cookies support?	True
VBScript support?	True
JavaScript support?	True

The Browscap.ini File

The "Browsercap.ini" file is used to declare properties and to set default values for browsers.

This section is not a tutorial on how to maintain "Browsercap.ini" files, it only shows you the basics; so you get an idea what a "Browsercap.ini" file is all about.

The "Browsercap.ini" file can contain the following:

```
[;comments]
[HTTPUserAgentHeader]
[parent=browserDefinition]
[property1=value1]
[propertyN=valueN]
[Default Browser Capability Settings]
[defaultProperty1=defaultValue1]
[defaultPropertyN=defaultValueN]
```

Parameter	**Description**
comments	Optional. Any line that starts with a semicolon are ignored by the BrowserType object
HTTPUserAgentHeader	Optional. Specifies the HTTP User Agent header to associate with the browser-property value statements specified in propertyN. Wildcard characters are allowed
browserDefinition	Optional. Specifies the HTTP User Agent header-string of a browser to use as the

	parent browser. The current browser's definition will inherit all of the property values declared in the parent browser's definition
propertyN	Optional. Specifies the browser properties. The following table lists some possible properties: • Backgroundsounds - Support background sounds? • Cdf - Support Channel Definition Format for Webcasting? • Tables - Support tables? • Cookies - Support cookies? • Frames - Support frames? • Javaapplets - Support Java applets? • Javascript - Supports JScript? • Vbscript - Supports VBScript? • Browser - Specifies the name of the browser • Beta - Is the browser beta software? • Platform - Specifies the platform that the browser runs on • Version - Specifies the version number of the browser
valueN	Optional. Specifies the value of propertyN. Can be a string, an integer (prefix with #), or a Boolean value
defaultPropertyN	Optional. Specifies the name of the browser property to which to assign a default value if none of the defined HTTPUserAgentHeader values match the HTTP User Agent header sent by the browser
defaultValueN	Optional. Specifies the value of defaultPropertyN. Can be a string, an integer (prefix with #), or a Boolean value

A "Browsercap.ini" file might look something like this:

```
;IE 5.0
[IE 5.0]
browser=IE
Version=5.0
majorver=#5
minorver=#0
frames=TRUE
tables=TRUE
cookies=TRUE
backgroundsounds=TRUE
vbscript=TRUE
javascript=TRUE
javaapplets=TRUE
ActiveXControls=TRUE
```

```
beta=False

;DEFAULT BROWSER
[*]
browser=Default
frames=FALSE
tables=TRUE
cookies=FALSE
backgroundsounds=FALSE
vbscript=FALSE
javascript=FALSE
```

Email Handling Using ASP

CDOSYS(CDO) is a built-in component in ASP. This component is used to send e-mails with ASP.

CDO (Collaboration Data Objects) is a Microsoft technology that is designed to simplify the creation of messaging applications.CDOSYS is a built-in component in ASP.

Examples using CDOSYS

Sending a text e-mail:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.TextBody="This is a message."
myMail.Send
set myMail=nothing
%>
```

Sending a text e-mail with Bcc and CC fields:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.Bcc="someoneelse@somedomain.com"
myMail.Cc="someoneelse2@somedomain.com"
myMail.TextBody="This is a message."
myMail.Send
set myMail=nothing
%>
```

Sending an HTML e-mail:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.HTMLBody = "<h1>This is a message.</h1>"
myMail.Send
set myMail=nothing
%>
```

Sending an HTML e-mail that sends a webpage from a website:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.CreateMHTMLBody "http://www.w3schools.com/asp/"
myMail.Send
set myMail=nothing
%>
```

Sending an HTML e-mail that sends a webpage from a file on your computer:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.CreateMHTMLBody "file://c:/mydocuments/test.htm"
myMail.Send
set myMail=nothing
%>
```

Sending a text e-mail with an Attachment:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.TextBody="This is a message."
myMail.AddAttachment "c:\mydocuments\test.txt"
myMail.Send
set myMail=nothing
%>
```

Sending a text e-mail using a remote server:

```
<%
Set myMail=CreateObject("CDO.Message")
myMail.Subject="Sending email with CDO"
myMail.From="mymail@mydomain.com"
myMail.To="someone@somedomain.com"
myMail.TextBody="This is a message."
myMail.Configuration.Fields.Item("http://schemas.microsoft.com/cdo/configuration/sendusing")=2
'Name or IP of remote SMTP server
myMail.Configuration.Fields.Item("http://schemas.microsoft.com/cdo/configuration/smtpserver")="smtp.
server.com"
'Server port
myMail.Configuration.Fields.Item("http://schemas.microsoft.com/cdo/configuration/smtpserverport")=25
myMail.Configuration.Fields.Update
myMail.Send
set myMail=nothing
%>
```

Unit 5: Accessing Databases with ASP and ADO

Active Database Object(ADO)

ADO represents a collection of objects that, via ASP, you can easily manipulate to gain incredible control over the information stored in your data source (be it an Access database, an Excel spreadsheet, and so on).

ADO is a Microsoft technology

- ADO stands for **A**ctiveX **D**ata **O**bjects
- ADO is a Microsoft Active-X component
- ADO is automatically installed with Microsoft IIS
- ADO is a programming interface to access data in a database

Within ADO are three major objects.

Connection Object

The ADO Connection Object is used to create an open connection to a data source. Through this connection, you can access and manipulate a database.

If you want to access a database multiple times, you should establish a connection using the Connection object. You can also make a connection to a database by passing a connection string via a Command or Recordset object. However, this type of connection is only good for one specific, single query.

```
set objConnection=Server.CreateObject("ADODB.connection")
```

Properties

Property	Description
ConnectionString	Sets or returns the details used to create a connection to a data source
ConnectionTimeout	Sets or returns the number of seconds to wait for a connection to open
Provider	Sets or returns the provider name
State	Returns a value describing if the connection is open or closed
Version	Returns the ADO version number

Methods

Method	Description
BeginTrans	Begins a new transaction
Cancel	Cancels an execution
Close	Closes a connection
CommitTrans	Saves any changes and ends the current transaction
Execute	Executes a query, statement, procedure or provider specific text
Open	Opens a connection
RollbackTrans	Cancels any transaction

Command Object

The ADO Command object is used to execute a single query against a database. The query can perform actions like creating, adding, retrieving, deleting or updating records.

If the query is used to retrieve data, the data will be returned as a RecordSet object. This means that the retrieved data can be manipulated by properties, collections, methods, and events of the Recordset object.

The major feature of the Command object is the ability to use stored queries and procedures with parameters.

```
set objCommand=Server.CreateObject("ADODB.command")
```

Properties

Property	Description
ActiveConnection	Sets or returns a definition for a connection if the connection is closed, or the current Connection object if the connection is open
CommandText	Sets or returns a provider command
CommandType	Sets or returns the type of a Command object
Name	Sets or returns the name of a Command object
State	Returns a value that describes if the Command object is open, closed, connecting, executing or retrieving data

Methods

Method	Description
Cancel	Cancels an execution of a method
CreateParameter	Creates a new Parameter object
Execute	Executes the query, SQL statement or procedure in the CommandText property

Collections

Collection	Description
Parameters	Contains all the Parameter objects of a Command Object

Recordset Object

The ADO Recordset object is used to hold a set of records from a database table. A Recordset object consist of records and columns (fields).

In ADO, this object is the most important and the one used most often to manipulate data from a database.

```
set objRecordset=Server.CreateObject("ADODB.recordset")
```

When you first open a Recordset, the current record pointer will point to the first record and the BOF and EOF properties are False. If there are no records, the BOF and EOF property are True.

Properties

Property	Description
ActiveCommand	Returns the Command object associated with the Recordset
ActiveConnection	Sets or returns a definition for a connection if the connection is closed, or the current Connection object if the connection is open
BOF	Returns true if the current record position is before the first record, otherwise false
DataSource	Specifies an object containing data to be represented as a Recordset object
EOF	Returns true if the current record position is after the last record, otherwise false
Filter	Sets or returns a filter for the data in a Recordset object
Index	Sets or returns the name of the current index for a Recordset object
LockType	Sets or returns a value that specifies the type of locking when editing a record in a Recordset
MaxRecords	Sets or returns the maximum number of records to return to a Recordset object from a query
RecordCount	Returns the number of records in a Recordset object
Sort	Sets or returns the field names in the Recordset to sort on
Source	Sets a string value or a Command object reference, or returns a String value that indicates the data source of the Recordset object
State	Returns a value that describes if the Recordset object is open, closed, connecting, executing or retrieving data

Methods

Method	Description
AddNew	Creates a new record
Cancel	Cancels an execution
CancelUpdate	Cancels changes made to a record of a Recordset object
Clone	Creates a duplicate of an existing Recordset
Close	Closes a Recordset
Delete	Deletes a record or a group of records
Find	Searches for a record in a Recordset that satisfies a specified criteria
GetRows	Copies multiple records from a Recordset object into a two-dimensional array
GetString	Returns a Recordset as a string
Move	Moves the record pointer in a Recordset object
MoveFirst	Moves the record pointer to the first record
MoveLast	Moves the record pointer to the last record
MoveNext	Moves the record pointer to the next record

MovePrevious	Moves the record pointer to the previous record
Open	Opens a database element that gives you access to records in a table, the results of a query, or to a saved Recordset
Save	Saves a Recordset object to a file or a Stream object

The common way to access a database from inside an ASP page is to:

1. Create an ADO connection to a database
2. Open the database connection
3. Create an ADO recordset
4. Open the recordset
5. Extract the data you need from the recordset
6. Close the recordset
7. Close the connection

```
<%
'declare the variable that will hold new connection object
Dim Connection
'create an ADO connection object
Set Connection=Server.CreateObject("ADODB.Connection")

'declare the variable that will hold the connection string
Dim ConnectionString
'define connection string, specify database driver and location of the database
ConnectionString="PROVIDER=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\inetpub\wwwroot\db\examples.mdb"

' Or Connection.Open "DSN=dsn_name"
'open the connection to the database
Connection.Open ConnectionString
%>
```

Now we have an active connection to our database. Let's retrieve all the records from the 'Cars' table. For that we have to create an instance of the recordset object and feed it an SQL statement.

```
<%
'declare the variable that will hold our new object
Dim Recordset
'create an ADO recordset object
Set Recordset=Server.CreateObject("ADODB.Recordset")

'declare the variable that will hold the SQL statement
Dim SQL
SQL="SELECT * FROM CARS"

'Open the recordset object executing the SQL statement and return records
Recordset.Open SQL, Connection
%>
```

We have returned a recordset based on our SQL statement so let's now print out them in the browser.

```
<%
'first of all determine whether there are any records
```

```
If Recordset.EOF Then
Response.Write("No records returned.")
Else
'if there are records then loop through the fields
Do While NOT Recordset.Eof
Response.write Recordset("Name")
Response.write Recordset("Year")
Response.write Recordset("Price")
Response.write "<br>"
Recordset.MoveNext
Loop
End If
%>
```

Finally, need to close the objects and free up resources on the server.

```
<%
Recordset.Close
Set Recordset=Nothing
Connection.Close
Set Connection=Nothing
%>
```

Let’s look at a sample script to get an idea how to connect to MS SQL Server database without DSN:

```
<%
'declare the variables
Dim Connection
Dim ConnString
Dim Recordset
Dim SQL

'define the connection string, specify database driver
ConnString="DRIVER={SQL Server};SERVER=localhost;UID=user1;PWD=password1;DATABASE=mydb;"

'declare the SQL statement that will query the database
SQL = "SELECT * FROM Students"

'create an instance of the ADO connection and recordset objects
Set Connection = Server.CreateObject("ADODB.Connection")
Set Recordset = Server.CreateObject("ADODB.Recordset")

'Open the connection to the database
Connection.Open ConnString

'Open the recordset object executing the SQL statement and return records
Recordset.Open SQL,Connection

'first of all determine whether there are any records
If Recordset.EOF Then
Response.Write("No records returned.")
Else
'if there are records then loop through the fields
Do While NOT Recordset.Eof
```

```
Response.write Recordset("name")
Response.write Recordset("address")
Response.write Recordset("phone")
Response.write "<br>"
Recordset.MoveNext
Loop
End If
'close the connection and recordset objects to free up resources
Recordset.Close
Set Recordset=nothing
Connection.Close
Set Connection=nothing
%>
```

Let’s look at a sample script to get an idea how to connect to MySQL database without DSN:

```
<%
'declare the variables
Dim Connection
Dim ConnectionString
Dim Recordset
Dim SQL

'declare the SQL statement that will query the database
SQL = "SELECT * FROM TABLE_NAME"

'define the connection string, specify database driver
ConnString = "DRIVER={MySQL ODBC 3.51 Driver}; SERVER=localhost; DATABASE=mydb;
UID=harry;PASSWORD=mypassword;"

'create an instance of the ADO connection and recordset objects
Set Connection = Server.CreateObject("ADODB.Connection")
Set Recordset = Server.CreateObject("ADODB.Recordset")

'Open the connection to the database
Connection.Open ConnString

'Open the recordset object executing the SQL statement and return records
Recordset.Open SQL,Connection

'first of all determine whether there are any records
If Recordset.EOF Then
Response.Write("No records returned.")
Else
'if there are records then loop through the fields
Do While NOT Recordset.Eof
Response.write Recordset("FIRST_FIELD_NAME")
Response.write Recordset("SECOND_FIELD_NAME")
Response.write Recordset("THIRD_FIELD_NAME")
Response.write "<br>"
Recordset.MoveNext
Loop
End If
```

```
'close the connection and recordset objects freeing up resources
Recordset.Close
Set Recordset=nothing
Connection.Close
Set Connection=nothing
%>
```

Insert

```
sql="INSERT INTO Students (name,address,phone) VALUES (“sam”,”kathmandu”,”9841242356”)
Connection.Execute(sql)
```

```
sql = “Update Students set name='john' where id=10”

Connection.Execute(sql)
```

Delete

```
sql="DELETE FROM Students where id=5"
```

Filter

The Filter property sets or returns a variant that contains a filter for the data in a Recordset object. The filter allows you to select records that fit a specific criteria.

When the Filter property is set, the cursor moves to the first record in the filtered Recordset. And, when the Filter property is cleared, the cursor moves to the first record in the unfiltered Recordset.

Examples of a criteria string:

- rs.Filter="name='Smith'"
- rs.Filter="name='Smith' AND Birthdate >= #4/10/70#"
- rs.Filter="Lastname='Jonson' OR Lastname='Johnson'"
- rs.Filter= "Lastname LIKE Jon*"
- rs.Filter = adFilterNone 'removes the filter'

Stored Procedures

A Stored Procedure is a prepared sql code that is saved into the database so that it can be reused over and over again. It is also possible to pass parameters to the stored procedure. A stored procedure can be a group of SQL statements compiled into a single execution plan.

Examples:

Stored Procedure for the Query : “Select * from Students”

```
Create PROCEDURE GetStudents

As BEGIN

 Select * from Students

End

GO
```

Stored procedure for insert sql statement

```
Create PROCEDURE InsertIntoStudents
        @name VARCHAR(15) = NULL,
        @address VARCHAR(20)=NULL,
        @phone VARCHAR(10) = NULL,
        @new_student_id INT OUTPUT
As
BEGIN
        Insert into Students(name,address,phone)values(@name,@address,@phone)
        Select @new_student_id = SCOPE_IDENTITY()
END
GO
```

Stored Procedure for the query to find a student whose name is “Harry”

```
Create PROCEDURE FindStudent
        @name VARCHAR(15)
AS
BEGIN
        Select * from students where name = @name
END
GO
```

Stored Procedures and Parameterized Query in ASP

```
Set con = Server.CreateObject(“ADODB.Connection”)
Set cmd = Server.CreateObject(“ADODB.Command”)
Set rs = Server.CreateObject(“ADODB.RecordSet”)
Constring = “Driver={Sql server};server=localhost;database=school;”
cmd.ActiveConnection = con
cmd.CommandText = “FindStudent”
cmd.CommandType = adCmdStoredProc
cmd.Parameters.Append cmd.CreateParameter(“@name”,advarchar,adparamInput,20,”Hari”)
rs = cmd.Execute
rs.open cmd,con
rs.close
con.close
set rs= nothing
set con = nothing
set cmd = nothing
```

Another Example

The Stored Procedure

```
CREATE PROCEDURE qryNumberOnCourse
@CourseID nvarchar(10), /* Input parameter */
@Number int OUTPUT /* Output parameter */
    AS
BEGIN
SELECT @Number=count(tblStudents.StudentID)
    FROM tblStudents INNER JOIN
    tblEnrolment ON
    tblStudents.StudentID = tblEnrolment.StudentID
WHERE tblEnrolment.CourseID = @CourseID
END
GO
```

The purpose of this stored procedure is to return the number of students who are enrolled on a particular course code. So the input parameter is the course code and the output parameter is the number of students.

The ASP Code

```
Dim cmd, rs, connect, intNumber

Set con = Server.CreateObject(“ADODB.Connection”)

Set cmd = Server.CreateObject ("ADODB.Command")
Constring = “Driver={Sql server};server=localhost;database=school;”

cmd.ActiveConnection = con
cmd.CommandText = "qryNumberOnCourse"
cmd.CommandType = adCmdStoredProc
cmd.Parameters.Append cmd.CreateParameter("@CourseID",adVarChar,adParamInput ,10,"C001")
cmd.Parameters.Append cmd.CreateParameter("@Number",adInteger,adParamOutput)
Set rs = cmd.Execute
intNumber = comm.Parameters("@Number")
set cmd = nothing
set con = nothing
%>
```

Set cmd = Server.CreateObject ("ADODB.Command")

This starts with creating a command object which provides us with the ability to execute our query.

Set con = Server.CreateObject(“ADODB.Connection”)

Constring = “Driver={Sql server};server=localhost;database=school;”

The connection string requires you to enter in your server name.

cmd.ActiveConnection = con

The active connection defines the connection to our database.

cmd.CommandText = "qryNumberOnCourse"

CommandText defines which command we are about to execute, which is our query.

cmd.CommandType = adCmdStoredProc

CommandType identifies the type of command being executed, which in this case is a stored procedure.

cmd.Parameters.Append cmd.CreateParameter("@CourseID",adVarChar,adParamInput ,10,"C001")

We must create a parameter object with the values we are using. In this example we are hard coding the value to be entered into the query which is C001.

cmd.Parameters.Append cmd.CreateParameter("@Number",adInteger,adParamOutput)

The output parameter is going to be passed to the variable @Number.

Set rs = cmd.Execute
intNumber = cmd.Parameters ("@Number")

Now we execute our parameter query and the result is placed into intNumber

set cmd = nothing
Response.Write (intNumber)

Finally, close and de-reference the objects and then display the variable intNumber.

Controlling Transactions In ASP

These 3 methods is used with the Connection object to save or cancel changes made to the data source.

BeginTrans

The BeginTrans method starts a new transaction.

CommitTrans

The CommitTrans method saves all changes made since the last BeginTrans method call, and ends the current transaction.

Since transactions can be nested, all lower-level transactions must be resolved before you can resolve higher-level transactions.

RollbackTrans

The RollbackTrans method cancels all changes made since the last BeginTrans method call, and ends the transaction.

ASP Application

As the Internet grew into a major player on the global economic front, so did the number of investors who were interested in its development. So, it may be wonder, how does the Internet continue to play a major role in communications, media and news? The key words are: Web Application Projects.

Web applications are business strategies and policies implemented on the Web through the use of User, Business and Data services. These tools are where the future lies.

Who Needs Web Applications and Why?

There are many entities that require applications for the Web-one example would be Business-to-Business interaction. Many companies in the world today demand to do business with each other over secure and private networks. This process is becoming increasingly popular with a lot of overseas companies who outsource projects to each other. From the simple process of transferring funds into a bank account, to deploying a large scale Web services network that updates pricing information globally, the adoption of a Web applications infrastructure is vital for many businesses.

The Web Application Model

The Web application model, like many software development models, is constructed upon 3 tiers: User Services, Business Services and Data Services. This model breaks an application into a network of consumers and suppliers of services.

The User Service tier creates a visual gateway for the consumer to interact with the application. This can range from basic HTML and DHTML to complex COM components and Java applets.

The user services then grab business logic and procedures from the Business Services. This tier can range from Web scripting in ASP/PHP/JSP to server side programming such as TCL, CORBA and PERL, that allows the user to perform complex actions through a Web interface.

The final tier is the Data Service layer. Data services store, retrieve and update information at a high level. Databases, file systems, and writeable media are all examples of Data storage and retrieval devices. For Web applications, however, databases are most practical. Databases allow developers to store, retrieve, add to, and update categorical information in a systematic and organized fashion.

Choosing the Right Project

Choosing the right types of projects to work on is an extremely important part of the Web application development plan.

Assessing to the resources, technical skills, and publishing capabilities should be first goal. Taking the 3 tiers into consideration, devise a list of all available resources that can be categorically assigned to each tier.

The next consideration should be the cost. Do we have a budget with which to complete this project? How much will it cost us to design, develop and deliver a complete project with a fair amount of success? These are questions that should be answered before signing any deals or contracts.

Let’s look at an example. A company called ABC needs to develop a Web application that will display sales information created by different sales agents. The data is updated daily through a completely automated process from all 3 service tiers. The client tells developer that this entire project must be done in ASP/SQL server and that developer should host the application as well.

After assessing all resources, developer and his/her team come to a conclusion that the company is unable to do data backups on a daily basis. After further discussion, s/he realizes that this is a

very important part of the setup for his/her client, and s/he should not risk taking a chance with the project. It's very likely that developer will be more prepared next time around, when a similar project lands on desk, so decline the job and recommend someone else who has the capabilities to do it right now.

The Phases in a Web Application Project

The Web application development process has 4 phases:

1. Envisioning the nature and direction of the project
2. Devising the plan
3. Development
4. Testing, support and stability

Let's look at each of these in more detail.

1. Envisioning the nature and direction of the project

In this phase, the management and developers assigned to the project come together and establish the goals that the solution must achieve. This includes recognizing the limitations that are placed on the project, scheduling, and versioning of the application. By the end of this phase, there should be clear documentation on what the application will achieve.

2. Devising the plan

In this phase, developer and developer team must determine the "how's" of the application.

What scripting language is most appropriate, which features must be included, and how long will it take? These are some of the questions that must be answered through this planning phase. The main tangents at this point are the project plan and functional specification. The project plan determines a timeframe of events and tasks, while the functional specification outlines in detail how the application will function and flow.

3. Development

Once the project plan and functional specification are ready, a baseline is set for the development work to begin. The programmer/s or Web developer/s begin coding, testing and publishing data. This phase establishes the data variables, entities and coding procedures that will be used throughout the remainder of the project. A milestone document is prepared by the development team, which is then handed to management for review.

4. Testing, support and stability

The stability phase of the application project mainly focuses on testing and the removal of bugs, discrepancies and network issues that may otherwise cause the application to fail. It is here that policies and procedures are established for a successful support system.

Knowing Options and Using them Wisely

Understanding of the architecture and procedures behind Web application development is like what technical options is needed to consider for the development process itself.

Windows Web Servers

Microsoft has built a loyal customer base on one important factor – their easy-to-use software. Windows NT/2000/XP Web servers are very fast and easy to administer. The fact that the operating system is a Windows shell means that administrators and authors can easily allow the Web server to interact with other software and hardware applications to transmit and receive data over the Internet. Popular server side scripting languages used with Windows servers are ASP/ASP. net, Java Server Pages, and PHP.

UNIX/Linux Web Servers

UNIX has long been known for its reliability. It is a powerful and robust Web server and operating system. Unix is the server of choice for many large-scale Websites that need content management systems or receive an extremely high volume of traffic. Popular server side scripting languages for UNIX are Java Server Pages, PERL, PHP, and CORBA

Every scripting language has its pros and cons. As I'm not writing a book here, I'll use the ASP model as my language of illustration. When working with Windows servers, there are several important parameters that the developer needs to throw into the equation, including security, scalability, speed and application design. So below I'm going to help you formulate a successful plan to accomplish all kinds of Web projects.

Planning for a Successful Web Development Project

In order to drastically minimize the risk of project failure, application development projects in the following sequence may be good approach.

1. Identify business logic and entities

Start by gathering information on everything. If we are going to be working with databases, begin by enumerating how many entities will be used in the business logic. For example, if program implements sales data, a sales ticket would be an entity.

Once all entities have been identified, establish a clear guideline for their relationships. This can be done via presentations, flowcharts or even reports.

2. Create a functional specification and project plan

This part is the most important part of the project. Functional specifications (or functional specs) are a map, or blueprint for how particular Web application looks and works. The spec details what the finished product will do, user interaction, and its look and feel.

An advantage of writing a functional spec is that it streamlines the development process. It takes discrepancies and guesswork out of the programming process, because the level of detail that goes into the plan makes it possible to minimize the misunderstanding that's usually associated with project mishaps.

Once the functional spec is finished, a project plan must be devised. A project plan is a timeline of tasks and events that will take place during the project. The project or program manager is normally the person who creates a project plan, and their primary focus is to detail task notes while being able to accommodate scheduling and resource information.

3. Bring the application model into play

As discussed earlier, the application model consists of 3 tiers – The User, Business and Data service tiers, each of which serves a substantial purpose.

Practically speaking, it's always best to start with the data tier, because entities and their relationships are already identified. The data tier can be an SQL server database, a text file, or even the powerful and robust Oracle. Create tables, relationships, jobs, and procedures depending on what platform chosen. If the data is a warehouse (i.e. the data already exists and does not depend on real time interaction), then make sure that new and additional data can be added securely and in a scalable fashion.

The Business services tier is the heart of the application. It involves the implementation of business logic into the scripting or programming language.

At this stage, make environment for testing and debugging has already been setup. Always test on at least 2 instances in application, after all, what may work perfectly, may not do so well on other platforms or machines. ASP, XML, PHP, JSP and CGI are some examples of server side

scripting languages used at the business service level. Whichever language choosen, make sure that it's capable of handling all the business logic presented in the functional specification.

The last is the user tier, which is absolutely vital for the interactive and strategic elements in the application. It provides the user with a visual gateway to the business service by placing images, icons, graphics and layout elements in strategic areas of interest, most commonly, based on management research.

4. Develop a support scheme

Being able to support and stabilize application is very important. Define a procedure call for cases of failure, mishaps or even downtime.

UNIT-7 ADVANCED ASP

.Net Framework

.NET is a software development and application execution environment that allows us to create, compile, test, deploy, and execute software that may be coded in a variety of different programming languages that adhere to a single set of Common Language Runtime files.

From a high-level view, the .NET Framework can be described as a little virtual operating system that runs on top of the operating systems.

The .NET Framework is an integral Windows component that supports building and running the next generation of applications. The .NET Framework is designed to fulfil the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.

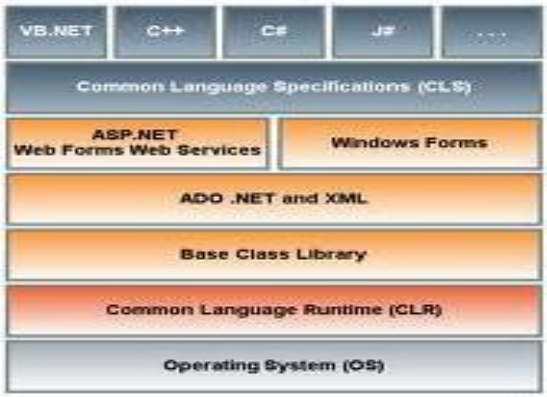


Figure 7.1 : .Net Architecture

.Net Architecture is explained from bottom to top in the following discussion.

1. At the bottom of the Architecture is **Common Language Runtime**. Dot NET Framework common language runtime resides on top of the operating system services. The common language runtime loads and executes code that targets the runtime. The runtime gives you, for example, the ability for cross-language integration.
2. .NET Framework provides a rich set of **class libraries**. These include base classes, like networking and input/output classes, a data class library for data access, and classes for use by programming tools, such as debugging services. All of them are brought together by the Services Framework, which sits on top of the common language runtime.
3. ADO.NET is Microsoft ActiveX Data Object (ADO) model for the .NET Framework. ADO.NET is not simply the migration of the popular ADO model to the managed environment but a completely new paradigm for data access and manipulation.

ADO.NET is intended specifically for developing web applications. This is evident from its two major design principles:

1. Disconnected Datasets In ADO.NET, almost all data manipulation is done outside the context of an open database connection.
2. Effortless Data Exchange with XMLDatasets can converse in the universal data format of the Web, namely XML.

4. The 4th layer of the framework consists of the Windows application model and, in parallel, the Web application model.
   The Web application model-in the slide presented as ASP.NET-includes Web Forms and Web Services.ASP.NET comes with built-in Web Forms controls, which are responsible for generating the user interface. They mirror typical HTML widgets like text boxes or buttons. If these controls do not fit your needs, you are free to create your own user controls.
   Web Services brings you a model to bind different applications over the Internet. This model is based on existing infrastructure and applications and is therefore standard-based, simple, and adaptable.
   Web Services are software solutions delivered via Internet to any device. Today, that means Web browsers on computers, for the most part, but the device-agnostic design of .NET will eliminate this limitation.
5. One of the obvious themes of .NET is unification and interoperability between various programming languages. In order to achieve this; certain rules must be laid and all the languages must follow these rules. In other words we cannot have languages running around creating their own extensions and their own fancy new data types. CLS is the collection of the rules and constraints that every language (that seeks to achieve .NET compatibility) must follow.
6. The CLR and the .NET Frameworks in general, however, are designed in such a way that code written in one language can not only seamlessly be used by another language. Hence ASP.NET can be programmed in any of the .NET compatible language whether it is VB.NET, C#, Managed C++ or JScript.NET/

The main two components of .NET Framework are: the common language runtime and the class library.

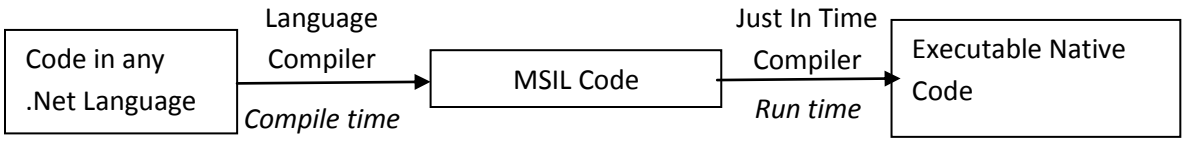
Common Language Runtime(CLR)

The common language runtime is the foundation of the .NET Framework. It can be considered as an agent that manages code at execution time, providing core services such as compilation, memory management, thread management, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code.

The runtime enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The common type system defines how types are declared, used, and managed in the runtime, and is also an important part of the runtime's support for cross-language integration. The CTS ensures that all managed code is self-describing. The various Microsoft and third-party language compilers generate managed code that conforms to the CTS. The common type system performs the following functions:

- Establishes a framework that helps enable cross-language integration, type safety, and high performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

When compilers emit code to run on the CLR, they do not emit machine language code. Rather, an intermediate language code is used called Microsoft Intermediate Language (MSIL). MSIL is like an object-oriented version of assembly language and is platform independent. It has a rich set of instructions that enable efficient representation of the code. When a code starts to execute, a process knowing as **Just in Time Compilation (JIT)** converts the MSIL code into the native processor instructions of the platform, which is then executed.



Just In Time compilers in many ways are different from traditional compilers as they compile the IL(intermediate Language) to native code only when desired; eg., when a function is called the Intermediate Language of the function's body is converted to native code just in time. So, part of code that is not used by the particular run is never converted to native code. If some IL code is converted to native code, then the next time it's needed the CLR reuses the same (already compiled) copy without re-compiling. So, if a program runs for some time (assuming that all or most of the functions get called), then it won't have any just in time performance penalty.

.Net FrameworkClass Library

The .NET Framework class library is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services. The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. Your collection classes will blend seamlessly with the classes in the .NET Framework.

The .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If you write an ASP.NET Web Form application, you can use the Web Forms classes.

.Net Languages

.Net framework supports many programming languages. Some of them are Visual Basic(VB), C#, C++, F# and others.

C# Language

C# is an elegant and type-safe object-oriented language that enables developers to build a wide range of secure and robust applications that run on the .NET Framework. You can use C# to create traditional Windows client applications, Web applications, client-server applications, database applications and much more. C# syntax is highly expressive, yet with less than 90 keywords, it is also simple and easy to learn.

It is a Case-Sensitive object-oriented language which supports the concepts of encapsulation, inheritance and polymorphism. All variables and methods, including the Main method, the application's entry point, are

encapsulated within class definitions. A class may inherit directly from one parent class, but it may implement any number of interfaces.

C# language is developed by Microsoft. The name "C sharp" was inspired by musical notation where a sharp indicates that the written note should be made a semitone higher in pitch.

Sample C# Program

```
//HelloWorld Program in c#
using System;
namespace HelloWorld
{
   class Hello
   {
      static void Main()
      {
         Console.WriteLine("Hello World!");

         // Keep the console window open in debug mode.
         Console.WriteLine("Press any key to exit.");
         Console.ReadKey();
      }
   }
}
```

```
using System;

namespace NamespaceReadString

{
        public class ReadString {
                public static void Main() {
                        string str;
                        Console.WriteLine("Enter some characters.");
                        str = Console.ReadLine();
                        Console.WriteLine("You entered: " + str);
                        Console.ReadKey();
                }
        }
}
```

Namespace in c#

Namespaces are C# program elements designed to help you organize your programs. A Namespace is simply a logical collection of related classes in C#. We bundle our related classes in some named collection calling it a namespace. As C# does not allow to classes with the same name to be used in a program, the sole purpose of using namespaces is to prevent name conflicts. For example, if you created a class named Console, you would need to put it in your own namespace, say MyNamespace, to ensure that there wasn't any confusion about when the System.Console class should be used or when your class should be used. To avoid this, these classes are made part of their respective namespace. So the fully qualified name of these classes will be MyNamespace.Console and System.Console, hence resolving any ambiguity for the compiler. The namespace

may contain classes, events, exceptions, delegates and even other namespaces called 'Internal namespace'. For Example.

```
namespace N1     // N1
{
   class C1      // N1.C1
   {
      class C2   // N1.C1.C2
      {
      }
   }
   namespace N2  // N1.N2
   {
      class C2   // N1.N2.C2
      {
      }
   }
}
```

Namespaces don't correspond to file or directory names. If naming directories and files to correspond to namespaces helps you organize your code, then you may do so, but it is not required.

The using Keyword

The using keyword allows us to use the classes in the following namespace. For example:
using System;
By doing this, we can now access all the classes defined in the System namespace like we are able to access the Console class in our Main method. using allows us to access the classes in the referenced namespace only and not in its internal/child namespace. Hence we might need to write
using System.Collections;
in order to access the classes defined in Collection namespace which is a sub/internal namespace of System namespace.

The Main method

A C# application must contain a Main method, in which control starts and ends. The Main method is where you create objects and execute other methods.The Main method is a static method that resides inside a class or a struct. The main method is the entry point of our program. The main method is designated as static as it will be called by the Common Language Runtime (CLR) without making any object.

In the previous "Hello World!" example, it resides in a class named Hello. You can declare the Main method in one of the following ways:

1>It can return void

```
static void Main()
{
   //...
}
```

2>It can also return an integer.

```
static int Main()
{
   //...
   return 0;
}
```

3> With either of the return types, it can take arguments.

```
static void Main(string[] args)
{
   //...
}
```
or
```
static int Main(string[] args)
{
   //...
   return 0;
}
```

Variables

During the execution of program, data is temporarily stored in memory. A variable is the name given to a memory location holding particular type of data. Each variable is associated with a data type and value.
<data_type> <variable>;
Eg: int i;
string myName;

Constants

Constants once defined, the value cannot be changed by program. All the constants are replaced by their values at the compile time.
Eg. const double PI 3.1416

Operators in C#
C# provides a large set of operators, which are symbols that specify which operations to perform in an expression.
Arithmetic operators: +,-,*,/,%,++,--
Assignment operators: =,+=,…
Relational operators: ==,!=,<,>,<=,>=
Logical operators: &, |, ^, !, ||

Example:
```
static void Main(string[] args)
{
        int num1 =  int.parse(args[0]);
        int num2 =  int.parse(args[1]);
        int sum =  num1+num2;
        Console.WriteLine("Sum of {0} and {1} is {2}.",num1,num2,sum);
        Console.WriteLine("Sum of "+ num1+ "and" + num2 + "is" + sum);
}
```

Blocks

The C#, Java, and C languages all rely heavily on curly braces—parentheses with a little more attitude: {}.Curly braces group multiple code statements together. Typically, the reason you'll want to group code statements together is because you want them to be repeated in a loop, executed conditionally, or grouped into a function.

```
{
// Code statements go here.
}
```

Loops

Looping statements repeat a specified block of code until a given condition is met.

For loop

```
for (int i = 0; i<=9; i++)
{
   System.Console.WriteLine(i);
}
```

foreach Loops

```
static void Main()
{
   string[] arr= new string[] {"Jan", "Feb", "Mar"};

   foreach (string s in arr)
   {
      System.Console.WriteLine(s);
   }
}
```

while and do...while Loops

```
while (condition)
{
   // statements
}
do
{
   // statements
}
while(condition);  // Don't forget the trailing ; in do...while loops
```

Conditional Logic

If else

```
if (myNumber > 10)
{
        // Do something.
}
else if (myString == "hello")
```

```
{
        // Do something.
}
else
{
        // Do something.
}
```

switch

C# also provides a switch statement that you can use to evaluate a single variable or expression for multiple possible values. The only limitation is that the variable you're evaluating must be an integer-based data type, a bool, a char, a string, or a value from an enumeration.

In the following code, each case examines the myNumber variable and tests whether it's equal to a specific integer:

```
switch (myNumber)
{
        case 1:
                // Do something.
                break;
        case 2:
                // Do something.
                break;
        default:
                // Do something.
                break;
}
```

Arrays and String

Array is the collection of similar data type. Each individual value in the array is accessed using one or more index numbers. It's often convenient to picture arrays as lists of data (if the array has one dimension) or grids of data (if the array has two dimensions). Typically, arrays are laid out contiguously in memory. All arrays start at a fixed lower bound of 0. When you create an array in C#, you specify the number of elements. Because counting starts at 0, the highest index is actually one fewer than the number of elements. (In other words, if you have three elements, the highest index is 2.)

Declaration

Single Dimension Array

<data_type>[ ] <identifier> = new <data_type> [<size>]

```
int[] numbers = new int[5];
or
int[] numbers;
numbers = new int[5];

string[] towns = new string[5];
int[] nums = {1,2,3,4,5};
string[] firstNames = {"hari","shyam", "michael"};
```

Multidimension Array

<data_type>[,] <identifier> = new <data_type> [<size>,<size>]

string[,] names = new string[2,4];
string[,] names = {{“hari”,”khadka”}, {“ram”, “pandey”}}
value of names[0,0] is “hari”, names[0,1] is “khadka”, names[1,1] is “pandey”
int[,] heightWidth = new int[5,2];
int[,] twoNums = {{1,2},{3,4},{5,6}};
value of twoNums[0,1] is 2.

Declaring arrays does not actually create the arrays. In C#, arrays are objects and must be instantiated.

foreach
foreach statement provides a simple, clean way to iterate through the elements of an array.

```
foreach(<type_of_element> <identifier> in <array>)
{
        <statement or  set of statements>
}
foreach(int i in numbers)
{
        Console.WriteLine(i);
}
```

In foreach, variable used to hold the individual elements of array in each iteration is read only. The elements in array cannot be changed through it. This means that foreach will only allow to iterate through the array or collection and not to change the contents of it. foreach can be used to iterate through arrays or collections. By collection, we mean any class, struct or interface that implements the IEnumerable interface.
String is also the collection of characters. So you can iterate over a string using **foreach**.

```
String firstName = “Michael”;
Foreach(char ch in firstName)
{
        Console.WriteLine(ch);
}
```

Class and Objects

A class is a construct that enables us to create your own custom types by grouping together variables of other types, methods and events. Together, these elements are called class members. A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on data (methods) and accessors to data(properties).
For example, there is a class String in the System namespace of .Net framework Class Library. This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like ToLowerCase(), Trim(), Substring() etc. It also has some propertied like Length.

An object is the concrete realization of instance built on the model specified by the class. An object is created in the memory using the keyword ‘new’ and is referenced by an identifier called a “reference”.
For example: MyClass myClassObject=new MyClass();
In this example, an object of MyClass is referenced by an identifier myClassObject .

Fields are the data contained in the class. Fields may be implicit data types, objects of some other class, enumerations, structs or delegates.

Methods are the operations performed on the data. A method may take some input values through its parameters and may return a value of particular data type.

A class can be static or non-static.

Static Class and Static Members

A static class is a class which cannot be instantiated. Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class. Static class members can be used to separate data and behavior that is independent of any object identity i.e. the data and functions do not change regardless of what happens to the object.

The main features of a static class are:

- They only contain static members.
- They cannot be instantiated.
- They are sealed.
- They cannot contain instance constructors but can have a static constructor

Keyword static is used to define a static class and static class members.

Example

```
static class CompanyInfo
{
   public static string GetCompanyName() { return "CompanyName"; }
   public static string GetCompanyAddress() { return "CompanyAddress"; }
   //...
}
```

Access Modifiers

Access modifiers are keywords used to specify the declared accessibility of a member or a type. C# has introduced the four access modifiers: public, protected, internal, and private.
The following five accessibility levels can be specified using the access modifiers:
public: Access is not restricted.
protected: Access is limited to the containing class or types derived from the containing class.
Internal: Access is limited to the current project(Assembly).
protected internal: Access is limited to the current project(Assembly) or types derived from the containing class.
private: Access is limited to the containing type.
Example:

```
using System;
namespace StudentNamespace
{
        public class student
        {
                //fields
                private int age,markMaths,markSci,markEng,obt;
                private string name;
                private double percentage;

                //methods
```

```
                public Void CalulateTotal()
                {
                        obt=markMaths+markSci+marking;
                }
                public Void CalculatePercentage()
                {
                        percentage=(double) obt/200*100;
                }
                public double GetPecentage
{
                return percentage;
}
//Main Methods
public Staic void Main()
{
                Student std=new Student();
                std.name="Ram".
                std.age=21;
                std.markMath=80;
                std.markSci=80;
                std,marking=80;
                std.CalculateTotal();
                std.CalculatePercentage();
                double per=std.GetPercentage();
}
        }
}
```

Properties

Properties are members that provide a flexible mechanism to read, write, or compute the values of private fields. Properties can be used as though they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily while still providing the safety and flexibility of methods.

Since all the fields of class are made private, to assign values to them through their reference 'Properties' is used. C# is the first language to provide the support of defining properties in the languages core. Properties hold specific information relevant to that class of object. Properties can be thought as characteristics of the objects that they represent. In traditional languages like java and C++, for accessing the private fields of class, public methods called getters (to retrieve the value) and setters (to assign the value) were defined like if we have a private field name:
private string name;
then the getters and setters would be like

```
//getter to name field
public string GetName()
{
        return name;
}
//setter to name field
public void SetName(string theName)
{
        name=theName;
}
```

These procedures gives a lot of control over how fields of classes should be accessed and dealt in a program. But, the problem is that two methods have to be defined and have to prefix the name of fields with Get or Set. C# provides the built in support for these getters and setters in the form of properties. Properties are context sensitive constructs used to read, write or compute private fields of class land to achieve control over how the fields can be accessed. The general syntax for properties is

```
<access_modifier> <data_type> <name_of_property>
{
        get
        {
                //some optional statements
                return <some_private_fields>;
        }
        set
        {
                //some optional statements
                <some_private_fields>=value;
        }
}
```

Example:

```
class TimePeriod
{
   private double seconds;

   public double Hours
   {
      get { return seconds / 3600; }
      set { seconds = value * 3600; }
   }
}

class Program
{
   static void Main()
   {
      TimePeriod t = new TimePeriod();

      // Assigning the Hours property causes the 'set' accessor to be called.
      t.Hours = 24;

      // Evaluating the Hours property causes the 'get' accessor to be called.
      System.Console.WriteLine("Time in hours: " + t.Hours);
   }
}
```

Auto-Implemented Properties

Auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors.

```
class Customer
{
```

```
    // Auto-Impl Properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    // Constructor
    public Customer(double purchases, string name)
    {
        TotalPurchases = purchases;
        Name = name;
    }
    // Methods
    public string GetContactInfo() {return "ContactInfo";}
    public string GetTransactionHistory() {return "History";}
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer ( 4987.63, "Northwind");

        //Modify a property
        cust1.TotalPurchases += 499.99;
    }
}
```

Constructors

Constructors are class methods that are executed when an object of a class or struct is created. They have the same name as the class or struct, and usually initialize the data members of the new object. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

A constructor has the following properties:
- It has the same name as its containing class.
- It has no return type.
- It is automatically called when a new instance or object of a class is created, hence why it's called a constructor.
- the constructor contains initialization code for each object, like assigning default values to the fields.

If you do not provide a constructor for your object, C# will create one by default that instantiates the object and sets member variables to the default values.

In the following example, a class named Taxi is defined by using a simple constructor. This class is then instantiated with the new operator. The Taxi constructor is invoked by the new operator immediately after memory is allocated for the new object.

```
public class Taxi
{
    public bool isInitialized;
    public Taxi()
```

```
   {
      isInitialized = true;
   }
}

class TestTaxi
{
   static void Main()
   {
      Taxi t = new Taxi();
      Console.WriteLine(t.isInitialized);
   }
}
```

A constructor that takes no parameters is called a default constructor. Default constructors are invoked whenever an object is instantiated by using the new operator and no arguments are provided to new.

Overloading of Constructors and Methods

It is possible to have more than one method with the same name and return type but with different number and types of arguments (parameters). This is called method overloading. For example, in the following class MathUtil, we have defined two methods with the name plus. Both of them have same return type but the first one has two parameters whereas the second one has 3 parameters. When plus() is called, the compiler will decide, on the basis of the types and number of parameters being passed, which one of these two to actually call, for example, plus(2,3) calls the first one and plus(2,3,4) calls the second one.

```
Class MathUtil

{
        public static int plus(int n1, int n2)

        {
                return(n1+n2);
        }
        public static int plus(int n1, int n2, int n3)

        {
                return (n1 + n2 +n3);
        }
        Static void Main()
        {
          plus(2,3);
          plus(2,3,4);
        }
}
```

Methods are overloaded depending on the parameter list and not on the return type. The WriteLine() method of Console class in the System namespace has 19 different overloaded forms.

Similarly, constructors can be overload. For example

```
public class Employee
{
```

```
    public int salary;

    public Employee(int annualSalary)
    {
        salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        salary = weeklySalary * numberOfWeeks;
    }
}
```

If we create an object like: Employee emp1=new Employee(1000);
the first constructor will be called initializing the salary to 1000. If we create an object like
Employee emp2=new Employee(1000,5);
the second constructor will be called which takes two arguments to calculate the salary.
Overloading methods and constructors gives program a lot of flexibility and reduces a lot of complexity that would otherwise be produced if we had to use different name for these methods.

Passing Parameters

In C# variables are of two types: reference types and value types.

A reference type is a type which has as its value a reference to the appropriate data rather than the data itself. Array is a reference type. Similarly Class types, interface types, delegate types are all reference types.

A value type is a type which contains the actual data. Variables of a value type directly contain the data. Simple types such as float, int, char are all value types.

In C#, parameters can be passed either by value or by reference. Passing parameters by reference allows function members (methods, properties, constructors) to change the value of the parameters and have that change persist. To pass a parameter by reference, use the **ref** or **out** keyword.

1>Passing Value-type parameters

Value-type parameters can be passed by value or by reference.

a> Passing Value Types by Value
Passing a value-types variable by value to a method means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the variable. For example

```
using System;
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of myInt.
    {
        x *= x;
        Console.WriteLine("The value inside the method: {0}", x);
                //output : The value inside the method : 25
    }
```

```
   public static void Main()
   {
      int myInt = 5;
      Console.WriteLine("The value before calling the method: {0}",
        myInt);
            //output : The value before calling the method: 5
      SquareIt(myInt);   // Passing myInt by value.
      Console.WriteLine("The value after calling the method: {0}",
        myInt);
            //output : The value after calling the method: 5
   }
}
```

The variable myInt, being a value type, contains its data (the value 5). When SquareIt is invoked, the contents of myInt are copied into the parameter x, which is squared inside the method. In Main, however, the value of myInt is the same, before and after calling the SquareIt method. In fact, the change that takes place inside the method only affects the local variable x.

b> Passing Value Types by Reference

If you want the called method to change the value of the parameter, you have to pass it by reference, using the **ref** or **out** keyword. Using **ref** requires the variable must be initialized before passing it to the method by reference whereas it's not necessary to initialize the variable before passing it to the method if we are using **out. out** is generally used when a method has to return multiple values.

Example – Using **ref**

```
using System;
class PassingValByRef
{
   static void SquareIt(ref int x)
   // The parameter x is passed by reference.
   // Changes to x will affect the original value of myInt.
   {
      x *= x;
      Console.WriteLine("The value inside the method: {0}", x);
            //output: The value inside the method: 25
   }
   public static void Main()
   {
      int myInt = 5;
      Console.WriteLine("The value before calling the method: {0}",
        myInt);
            //output: The value before calling the method: 5
      SquareIt(ref myInt);   // Passing myInt by reference.
      Console.WriteLine("The value after calling the method: {0}",
        myInt);
            //Output: The value after calling the method: 25
   }
}
```

In this example, it is not the value of myInt that is passed; rather, a reference to myInt is passed. The parameter x is not an **int**; it is a reference to an **int** (in this case, a reference to myInt). Therefore, when x is squared inside the method, what actually gets squared is what x refers to: myInt.

The keyword out can be used similarly.

```
class OutExample
{
   static void Method(out int i)
   {
      i = 44;
   }
   static void Main()
   {
      int value;
      Method(out value);
      // value is now 44
   }
}
```

Swapping value types

A common example of changing the values of the passed parameters is the Swap method, where you pass two variables, x and y, and have the method swap their contents. You must pass the parameters to the Swap method by reference; otherwise, you will be dealing with a local copy of the parameters inside the method. The following is an example of the Swap method that uses reference parameters:

```
static void SwapByRef(ref int x, ref int y)
{
   int temp = x;
   x = y;
   y = temp;
}
```

When you call this method, use the **ref** keyword in the call, like this:

```
SwapByRef (ref i, ref j);
```

2>Passing Reference-Type Parameters

A variable of a reference type does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data pointed to by the reference, such as the value of a class member. However, you cannot change the value of the reference itself; that is, you cannot use the same reference to allocate memory for a new class and have it persist outside the block. To do that, pass the parameter using the **ref** (or **out**) keyword.

a> Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, myArray, by value, to a method, Change. Because the parameter is a reference to myArray, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, myArray.

```
// Passing an array to a method without the ref keyword.
using System;
class PassingRefByVal
{
```

```
  static void Change(int[] arr)
  {
    arr[0]=888;   // This change affects the original element.
    arr = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
    Console.WriteLine("Inside the method, the first element is: {0}", arr[0]);
    //Output:           Inside the method, the first element is: -3
  }

  public static void Main()
  {
    int[] myArray = {1,4,5};
    Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", myArray [0]);
          //Output: Inside Main, before calling the method, the first element is: 1
    Change(myArray);
    Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", myArray [0]);
//Output: Inside Main, after calling the method, the first element is: 888
  }
}
```

In the preceding example, the array, myArray, which is a reference type, is passed to the method without the **ref** parameter. In such a case, a copy of the reference, which points to myArray, is passed to the method. The output shows that it is possible for the method to change the contents of an array element (from 1 to 888). However, allocating a new portion of memory by using the **new** operator inside the Change method makes the variable arr reference a new array. Thus, any changes after that will not affect the original array, myArray, which is created inside Main. In fact, two arrays are created in this example, one inside Main and one inside the Change method.

b>Passing Reference Types by Reference

This example is the same as previous, except for using the **ref** keyword in the method call. Any changes that take place in the method will affect the original variables in the calling program.

```
using System;
class PassingRefByRef
{
  static void Change(ref int[] arr)
  {
    // Both of the following changes will affect the original variables:
    arr[0]=888;
    arr = new int[5] {-3, -1, -2, -3, -4};
    Console.WriteLine("Inside the method, the first element is: {0}", arr[0]);
//output : Inside the method, the first element is: -3
  }

  public static void Main()
  {
    int[] myArray = {1,4,5};
    Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", myArray [0]);
//Output : Inside Main, before calling the method, the first element is: 1
    Change(ref myArray);
    Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", myArray [0]);
//Output: Inside Main, after calling the method, the first element is: -3
  }
}
```

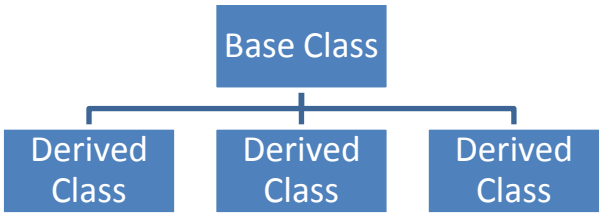
All of the changes that take place inside the method affect the original array in Main. In fact, the original array is reallocated using the **new** operator. Thus, after calling the Change method, any reference to myArray points to the five-element array, which is created in the Change method.

Inheritance

Inheritance enables you to create new classes that reuse, extend and modify the behavior that is defined in other class. The class whose members are inherited is called base or parent or super class and the class that inherits those members is called the derived or child or sub class.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors. The derived class can thereby reuse the code in the base class without having to re-implementing it. You can add more members in the derived class.

- A derived class has only one direct base class
- Inheritance is transitive
  class C ← class B ← class A
  class C ← class A
- Conceptually, a derived class is a specialization of the base class.
- System.Object is the ultimate base class for all classes in c#.



Inheritance can be done using '**:**' operator.

```
public class Animal
{
}
public class Mammal : Animal
```

```
{
}
public class Reptile : Animal
{
}
```

In the example above, class **Mammal** is effectively both **Mammal** and **Animal**. When you access a **Mammal** object, you can use the cast operation to convert it to an **Animal** object. The **Mammal** object is not changed by the cast, but your view of the **Mammal** object becomes restricted to **Animal**'s data and behaviors. After casting a **Mammal** to an **Animal**, that **Animal** can be cast back to a **Mammal**. Not all instances of **Animal** can be cast to **Mammal** —just those that are actually instances of **Mammal**. If you access class **Mammal** as a **Mammal** type, you get both the class **Animal** and class **Mammal** data and behaviors. The ability for an object to represent more than one type is called polymorphism.

Consider a class Student with the fields registrationNumber, name and dateOfBirth, along with the corresponding properties. The class has a method GetAge() which returns the age of student in years.

```
public class Student
{
   //fields
   private string name;
   private int registrationNumber;
   private DateTime dateOfBirth;
   private int age;

   //methods
   public int RegistrationNumber
   {
      get { return registrationNumber; }
      set { registrationNumber = value; }
   }
   public string Name
   {
      get{return name;}
      set{name=value}
   }
   public DateTime DOB
   {
      get { return dateOfBirth; }
      set { dateOfBirth = value; }
   }
   //constructors
   public Student()
   {
      System.Console.WriteLine("New student created without parameters");
   }
   public Student(int rNo, string nm, DateTime dob)
   {
      this.RegistrationNumber = rNo;
      this.Name = name;
      this.DOB = dob;
   }
   public int getAge()
   {
      int age = (DateTime.Now.Year - dateOfBirth.Year);
      return age;
   }
```

Now, Let us declare another class named SchoolStudent that inherits the Student class but with additional members like marks of different subjects and methods for calculating total marks and percentage.

```
public class SchoolStudent: Student
{
   private int totalMarks;
   private int obtMarks;

   //Constructors
   public SchoolStudent()
   {
      System.Console.WriteLine("New School Student create without parameters");
   }
   public SchoolStudent(int rNo, string name, DateTime dob, int totalMarks, int obtMarks, int CRN): base(rNo, name, dob)
   {
      this.totalMarks=totalMarks;
      this.obtMarks=obtMarks;
      this.classRollNumber=CRN;
   }
   public int ObtainedMarks
   {
      get{return obtMarks;}
      set{obtMarks=value;}
   }
   public int TotalMarks
   {
      get{return totalMarks;}
      set{totalMarks=value;}
   }
   public double GetPercentage()
   {
      percentage=(double)obtMarks/totalMarks*100;
            return percentage;
   }
}
```

The SchoolStudent class inherits the Student class by using the colon operator. The SchoolStudent class inherits all the members of the Student class. In additions, it also declares its own members: two private fields with their corresponding properties, two constructors and one instance method.

Constructor calls in inheritance

When we instantiate the sub-class (SchoolStudent), the compiler first instantiates the base class (Student) by calling one of its constructor and then calling the constructor of the sub-class. We can explicitly call the constructor of the base class using the keyword *base*. *base* should be used with the constructor heading (or signature or declaration) after a colon (:).

```
public class SubClass : BaseClass
{
   SubClass(int id): base()
   {
      // statements
   }
}
```

In code above, the parameterized constructor of the sub-class (SubClass) is explicitly calling the parameterless constructor of the base class. We can also calls the parameterized of the base-class through the base keyboard like:

```
public class SubClass :BaseClass
{
        SubClass(int id) :base (id) //explicit constructor call
        {
                //statements
        }
}
```

Now, the constructor of SubClass(int) will explicitly call the constructor of the base-class (BaseClass) that takes an int argument. In our SchoolStudent Class,

```
public SchoolStudent(int regNum, string name, DateTime dob, int totalMarks, int obtMarks): base (roll,name,dob)
{
  this.totalMarks = totalMarks;
  this.obtMarks = obtMarks;
}
```

The constructor above calls the parameterized constructor of its base-class(Student) , delegating the initialization of the inherited fields to the base-class's parameterized constructor.

```
public Student(int registrationNumber, string name, DateTime dateOfBirth
{
 this. registrationNumber = registrationNumber;
 this.Name=name
 this.dob = dob;
}
```

Sealed Class

Sealed classes are used to prevent a class from being used as a base class. It is primarily useful to prevent unintended derivations.

```
sealed class TestClass
{
   public TestClass()
   {
   }
}
class NewClass: TestClass //fails
{
}
```

Sealed Method

A class member, method, field, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the **sealed** keyword before the override keyword in the class member declaration. For example:

```
Public class C { public virtual void DoWork(){}; }
```

```
public class D : C { public sealed override void DoWork() { } }
```

Abstract Class

When we want to define a base class but do not want to instantiate it, then we define abstract class using the keyword abstract. Other classes can be derived from abstract classes. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract method does not have to contain abstract members, but if a class contains an abstract member, the class itself must be declared as abstract. Derived classes of the abstract class must implement all abstract methods. Abstract classes are useful when creating components because they allow you specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. They also version well, because if additional functionality is needed in derived classes, it can be added to the base class without breaking code.

```
public abstract Class A
{
   public abstract void DoWork(int i);
}
```

When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.

```
public abstract Class D
{
   public virtual void DoWork(int i)
   {
      // original implementation
   }
}
public abstract class E:D
{
   public abstract override void DoWork(int i);
}
public class F:E
{
   public override void DoWork(int i)
   {
      //New implementation
   }
}
```

In the following example, the class MyDerivedC is derived from an abstract class MyBaseC. The abstract class contains an abstract method, MyMethod(), and two abstract properties, GetX() and GetY().

```
// abstract_keyword.cs
// Abstract Classes
using System;
abstract class MyBaseC   // Abstract class
{
  protected int x = 100;
  protected int y = 150;
```

```
  public abstract void MyMethod();   // Abstract method

  public abstract int GetX   // Abstract property
  {
    get;
  }

  public abstract int GetY   // Abstract property
  {
    get;
  }
}

class MyDerivedC: MyBaseC
{
  public override void MyMethod()
  {
    x++;
    y++;
  }

  public override int GetX   // overriding property
  {
    get
    {
      return x+10;
    }
  }

  public override int GetY   // overriding property
  {
    get
    {
      return y+10;
    }
  }

  public static void Main()
  {
    MyDerivedC mC = new MyDerivedC();
    mC.MyMethod();
    Console.WriteLine("x = {0}, y = {1}", mC.GetX, mC.GetY);
  }
}
```

Polymorphism

Polymorphism is the ability to create a variable, a function, or an object that has more than one form. The word derives from the Greek word which has meaning "having multiple forms". It is the third pillar of object oriented programming after encapsulation after inheritance. Polymorphism is the ability for classes to provide different implementations of methods that are called by the same name. The primary usage of

polymorphism in object-oriented programming is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior. The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behavior is determined at run-time (this is called *late binding* or *dynamic binding*).

Through inheritance, a class can be used as more than one type; it can be used as its own type, any base types, or any interface type if it implements interfaces. This is called polymorphism. In C#, every type is polymorphic. Types can be used as their own type or as a Object instance, because any type automatically treats **Object** as a base type.

Polymorphism has two distinct aspects:

1>At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.

2>Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. To change the data and behavior of a base class, you have two choices: you can replace the base member with a new derived member, or you can override a virtual base member.

Replacing a member of a base class with a new derived member requires the **new** keyword. If a base class defines a method, field, or property, the **new** keyword is used to create a new definition of that method, field, or property on a derived class. The **new** keyword is placed before the return type of a class member that is being replaced. To override the virtual base member, the **override** keyword is used.

```
public class ClassA

{
   public void myMethod()
   {
      System.Console.WriteLine("Parent class method");
   }
}
public class ClassB : ClassA
{
   public void myMethod()
   {
      System.Console.WriteLine("Child Class Method");
   }
}
public class MyClass
{
   static void Main()
   {
      ClassB b = new ClassB();
      b.myMethod(); //output: child class method

      ClassA b1 = new ClassB();
      b1.myMethod(); //output: parent class method
```

```
    }
}
```

Overriding the methods - Virtual and Override Keywords

```
public class classA
{
    public virtual void Method()
    {
        System.Console.WriteLine("ClassA Method");
    }
}
public class classB : classA
{
    public override void Method()
    {
        System.Console.WriteLine("ClassB Method");
    }
}
public class MyClass
{
    static void Main()
    {
        classA a = new classB();
        a.Method(); //output: classB Method
    }
}
```

As we have overridden the method of classA in classB and since the method is marked virtual in classA, the compiler will no longer see the apparent (or reference) type to call the static early or compile time object binding rather it will apply dynamic, late or runtime object binding and will see the object type at the runtime to decide which method it should call. This procedure is called polymorphism, where we have different implementations of a method with the same name and signature in the base-class and sub-classes.

When such a method is called using a base-type-reference, the compiler uses the actual object type reference by the base type reference to decide which of the method to call.

By marking *virtual*, we allow a method to be overridden and be used polymorphically. In the same way, we make the method in the sub-class as override when it is overriding the virtual method in the base class.

Hiding base class members with new members

The *new* keyword:

If you want to define a member in the base class with same name but do not want to override the base class method, you can use the *new* keyword. The new modifier instructs the compiler to use your implementation instead of the base class implementation. Any code that is not referencing your class but the base class will use the base class implementation.

```
public class BaseClass
{
    public int num = 0;
```

```
    public void DoWork()
    {
        num++;
    }
    public int WorkProperty
    {
        get { return 0; }
    }
    public class DerivedClass : BaseClass
    {
        public new int num = 20;
        public new void DoWork()
        {
        }
        public new int WorkProperty
        {
            get { return 0; }
        }
    }
    static void Main()
    {
        DerivedClass B = new DerivedClass();
        B.DoWork(); //calls the new method
        BaseClass A = (BaseClass)B;
        A.DoWork(); //calls the old method
    }
}
```

Override and new keyword usage

C# enables methods in derived classes to have the same name as methods in base classes—as long as you are very specific about how the new method should be treated. The following example demonstrates the use of the new and override keywords.

First we declare three classes: a base class called Car, and two classes that derive from it. ConvertibleCar and Minivan. The base class contains a single method,DescribeCar, which sends a description of the car to the console. The derived class methods also include a method called DescribeCar, which displays their unique properties. These methods also call the base class DescribeCar method to demonstrate how they have inherited the properties of the Car class.

In order to highlight the difference, the ConvertibleCar class is defined with the **new** keyword, while the Minivan class is defined with **override**.

```
// Define the base class

class Car

{
    public virtual void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
    }
}

// Define the derived classes
```

```
class ConvertibleCar : Car
{
   public new virtual void DescribeCar()
   {
      base.DescribeCar();
      System.Console.WriteLine("A roof that opens up.");
   }
}

class Minivan : Car
{
   public override void DescribeCar()
   {
      base.DescribeCar();
      System.Console.WriteLine("Carries seven people.");
   }
}
```

We can now write some code that declares instances of these classes, and calls their methods so that the objects can describe themselves:

```
public static void TestCars1()
{
   Car car1 = new Car();
   car1.DescribeCar();
   System.Console.WriteLine("----------");

   ConvertibleCar car2 = new ConvertibleCar();
   car2.DescribeCar();
   System.Console.WriteLine("----------");

   Minivan car3 = new Minivan();
   car3.DescribeCar();
   System.Console.WriteLine("----------");
}
```

As you might expect, the output looks like this:

Four wheels and an engine.

Four wheels and an engine.

A roof that opens up.

Four wheels and an engine.

Carries seven people.

However, in this next section of code, we declare an array of objects derived from the Car base class. This array can store Car, ConvertibleCar, and Minivanobjects. The array is declared like this:

```
public static void TestCars2()

{
   Car[] cars = new Car[3];
   cars[0] = new Car();
   cars[1] = new ConvertibleCar();
   cars[2] = new Minivan();
}
```

We can then use a **foreach** loop to visit each Car object contained in the array, and call the DescribeCar method, like this:

```
foreach (Car vehicle in cars)
{
   System.Console.WriteLine("Car object: " + vehicle.GetType());
   vehicle.DescribeCar();
   System.Console.WriteLine("----------");
}
```

The output from this loop is as follows:

```
Car object: YourApplication.Car

Four wheels and an engine.

----------

Car object: YourApplication.ConvertibleCar

Four wheels and an engine.

----------

Car object: YourApplication.Minivan

Four wheels and an engine.

Carries seven people.

----------
```

Notice how the ConvertibleCar description is not what you might expect. As the **new** keyword was used to define this method, the derived class method is not called—the base class method is called instead.
The Minivan object correctly calls the overridden method, producing the results we expected.

If you want to enforce a rule that all classes derived from Car must implement the DescribeCar method, you should create a new base class that defines the method DescribeCar as **abstract**. An abstract method does not contain any code, only the method signature. Any classes derived from this base class must provide an implementation of DescribeCar.

override extends the base class method, whereas *new* keyword hides the base class method.

Interfaces

Interfaces are a special kind of type in c# used to define the specifications that should be followed by its sub-types. An interface can be defined using the *interface* keyword.
An interface has the following properties:

1. An interface is like an abstract class which can not be instantiated.
2. Any class implementing the interface must implement all of its members.
3. Interface can contain signatures for methods, properties, events.
4. Interface contain no implementation of methods
5. Classes can implement more than one interface
6. An interface itself can inherit from multiple interfaces.
7. Interface can not contain constants, fields, instance constructors, destructors or types
8. All members are public and abstract by default

Interfaces provide a way to achieve runtime polymorphism.

```
interface ISampleInterface
{
   void SampleMethod();
}
class TestClass : ISampleInterface
{
   public void SampleMethod()
   {
      System.Console.WriteLine("Interface method implementation");
   }
   static void Main()
   {
      ISampleInterface obj = new TestClass();
      obj.SampleMethod();
   }
}
```

The following example demonstrates interface implementation. In this example, the interface IPoint contains the property declaration, which is responsible for setting and getting the values of the fields. The class Point contains the property implementation.

```
using System;
interface IPoint
{
   //property signatures
   int x
   { get; set; }
   int y
   { get; set; }
}
class Point : IPoint
{
   //fields
   private int _x;
   private int _y;
   //constructor
   public Point(int x, int y)
   {
      _x = x;
      _y = y;
```

```
    }
    //property implementation
    public int x
    {
        get { return _x; }
        set { _x = value; }
    }
    public int y
    {
        get { return _y; }
        set { _y = value; }
    }
}
class MainClass
{
    static void PrintPoint(IPoint P)
    {
        Console.WriteLine("X= {0}, y={1}", P.x, P.y);
    }
    static void Main()
    {
        Point P = new Point(2, 3);
        PrintPoint(P);
    }
}
```

Partial Class

It is possible to split the definition of a class or an interface over two or more source files. Each source file contains a section of the class definition, and all parts are combined when the application is compiled.

Benefits

1. More than one developer can simultaneously work on the same class
2. When working with automatically generated source code, code can be added to the class without having to re-create the source file. Visual studio uses this approach when it creates windows forms etc. You can create code that uses these classes without having to modify the file created by visual studio.

To split a class definition, use the partial keyword modifier

```
public partial class Employee
{
    public void DoWork()
    {
    }
}
public partial class Employee
{
    public void MyWork()
    {
    }
}
```

The partial keyword indicates that other parts of the class; interface can be defined in the namespace. All the parts must use the partial keyword. All the parts must be available at compile time to form the fixed type. All the parts must have same accessibility i.e. private

- if any part is declared abstract, then the whole type is considered abstract
- if any part is declared sealed, then the whole type is considered sealed
- if any part declares a base type, then the whole type inherits that class
- any class, struct or interface member declared in a partial definition are available to all other parts. The final type is the combination of all the parts at compile time.

Example:

```
public partial class Coordinates
{
   private int x;
   private int y;
   public Coordinates(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
}
public partial class Coordinates
{
   public void PrintCoordinates()
   {
      System.Console.WriteLine("Coordinates: {0},{1}", x, y);
   }
}
class TestCoordinates
{
   static void Main()
   {
      Coordinates mycords = new Coordinates(10, 15);
      mycords.PrintCoordinates();
   }
}
```

Interface example:

```
partial interface ITest
{
   void Interface_Test();
}
partial interface ITest
{
   void Interface_Test2();
}
```

Partial Method

file1.cs.

```
partial void onNameChanged();
```

file2.cs

```
partial void onNameChanged();
```

A partial class may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls are removed at compile time.

A partial method declaration consists of two parts: the definition, and the implementation. They may be in separate parts of the partial class, or in the same part.

Partial method should have
- partial keyword
- return type void
- implicitly private, cannot be virtual

Collections

Collections are the enumerable data structures that can be accessed using indexes or keys. Closely related data can be handled more efficiently when grouped together into a collection. Instead of writing separate code to handle each individual object. We can use the same code to process all the elements of a collection.

To manage a collection, the array class and the System.Collections classes are used to add, remove and modify either individual elements of the collection or a range of elements.

The collection classes provide support for stacks, queues, lists and hash tables. Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs.

Example:

```
ArrayList list=new ArrayList()
list.Add(10);
list.Add("John");
list.Add('R');
```

Any reference or value type that is added to an **ArrayList** is implicitly upcast to **Object**. If the items are value types, they must be boxed when added to the list, and unboxed when they are retrieved. Both the casting and the boxing and unboxing operations degrade performance; the effect of boxing and unboxing can be quite significant in scenarios where you must iterate over large collections.

Properties:
- collections are defined as part of the System.Collections or System.Collections.Generic namespace
- most collection derive from the interfaces ICollection, IComparer, IEnumerable, IList, IDictionary and IDictionaryEnumerator and their generic equivalents.

System.Collection classes
- Queue ⟶ FIFO
- Stack ⟶ LIFO
- Linked list ⟶ Sequential access

```
Stack mystack = new Stack();
mystack.push("Hello");
mystack.push(1);
foreach(object obj in mystack)
{
        System.Console.WriteLine({0}, obj);
}
```

.....

```
public class SampleQueue
{
   public static void Main()
   {
      Queue myQ = new Queue();
      myQ.Enqueue("Hello");
      myQ.Enqueue("world");
      myQ.Enqueue("!");
      myQ.Dequeue();
      System.Console.WriteLine(myQ.count);
      printValues(myQ);
   }
   public static void printValues(IEnumerable list)
   {
      foreach object obj in list)
      {
         System.Console.WriteLine(obj);
      }
   }
```

Stack

```
Stack myStack = new Stack();
myStack.Push("Hello");
myStack.Push("World");
myStack.Push("!");
System.Console.WriteLine(mystack.count);
```

Sorted List

Maintain the elements sorted based on the key.

```
SortedList mySL = new SortedList();
mySL.Add("Third", "!");
mySL.Add("Second", "World");
mySL.Add("First", "Hello");
System.Console.WriteLine(mySL.Count);
System.Console.WriteLine(mySL.Capacity);

PrintKeysAndValues(mySL);

public static void PrintKeysAndValues(SortedList myList)
{
        System.Console.WriteLine("\t-key-\t-value")
        for(int i=0;i<myList.Count; i++)
       {
        System.Console.WriteLine("\t{0} : \t{1}", myList.GetKey(i), myList.GetByIndex(i);
            System.Console.WriteLine();
       }
 }
```

Generics

Generic classes and methods combine re-usability, type safety and efficiency to their counterpart collections. Generics are most commonly used with collections and the method that operate on them. .NET Framework 2.0 class library provide a new namespace *System.Collections.Generic* which contains several new generic based collection classes. Generics are checked at compile time.
List is a generic form of a ArrayList.

```
List<string> names = new List<string>();
names.Add(“John”);
names.Add(“Danny”);
static void Main(string[] args)
foreach (string name in names)
{
        System.Console.WriteLine(name);
}
names.Insert(2, "Rafael");
names.Remove("Hari");
```

Generics

Queue

```
Queue<string> numbers = new Queue<string>();
numbers.Enqueue("One");
numbers.Enqueue("two");
numbers.Enqueue("three");
foreach(String number in numbers)
{
   Console.WriteLine(number);
}
numbers.Dequeue();

Queue<string> queueCopy = new Queue<string>(numbers.ToArray);
foreach(string number in queueCopy)
{
   Console.WriteLine(number);
}
string[] array1 = new string [numbers.count*2];
numbers.CopyTo(array2, numbers.count);

Queue<string> queueCopy2 = new Queue<string>(array)
foreach(string number indexer queueCopy2)
{
   Console.WriteLine(numbers);
}

queueCopy.Contains("Four"); //true
queueCopy.Clear();
Console.WriteLine(queueCopy.count); //0
```

Generic Sorted List

```
SortedList<string, string> OpenWith = new SortedList<string, string>();
OpenWith.Add("txt", "notepad.exe");
OpenWith.Add("bmp", "paint.exe");
OpenWith.Add("dib", "paint.exe");
```

```
OpenWith.Add("rtf", "wordpad.exe");
OpenWith.Add("txt", "notepad.exe"); //error
Console.Writeline("For Key = \"rtf\",value={0}",OpenWith["rtf"];
OpenWith["rtf"] =  "winword.exe";
OpenWith["doc"] =  "winword.exe";  //adds new element

foreach[keyValuePair<string, string> kvp indexer OpenWith)
{
   Console.WriteLine("Key = {0}, Value = {1}",kvp.Key, kvp.Value);
}

IList<string> iListValues = OpenWith.Values;
```

Custom Collection class

```
using System.Collections
public class Tokens: IEnumerable
{
   private string[] elements;
   Tokens(string source, char[] delimeters);
   {
      elements = source.Split(delimeters);
   }
   public IEnumerator GetEnumerator()
   {
      return new TokenEnumerator(this);
   }
}

//Inner class implements IEnumerator Interface

private class TokenEnumerator: IEnumerator
{
   private int position= -1;
   private Tokens t;
   private TokenEnumerator(Tokens t)
   {
      this.t=t;
   }
}
//Declare the MoveNext method required by IEnumerator
public bool MoveNext()
{
   if(position < t.elements.Length=1)
   {
      position ++; return true;
   }
   else
   {
      return false;
   }
}
public void Reset()
{
   position = -1;
}
// Declare the current property required by IEnumerator
public object Current
{
```

```
    get{return t.element[position];}
}
static void Main()
{
    Token f = new Tokens("This is a sample sentence ", new char[]{'_',',','_'});
    foreach(string item in f)
    {
        System.Console.WriteLine(item);
    }
}
```

Jagged Array

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is also called an array of arrays. Multidimensional array are of fixed length whereas Jagged array are of variable length. So, performance is good.

```
int [][] jaggedArray = new int[3][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is the single dimension array of integers.

```
jaggedArray[0] = new int[]{1,2,3,4};
jaggedArray[1] = new int[]{0,2,4,3};
jaggedArray[2] = new int[]{11,12};

int [][] jaggedArray = new int[][];
{
    new int[]{1,3,5,7,9},
    new int[]{0,2,4,6},
    new int[]{11,12}
};
jaggedArray[0][1] =50;
jaggedArray[2][1] =20;

int [][] jaggedArray = new int[3][];
{
    new int[,]{{1,3},{5,7}},
    new int[,]{{0,2},{4,6},{8,10}},
    new int[,]{{11,12},{99,88},{0,9}}
}

jaggedArray4[0][1,0];
```

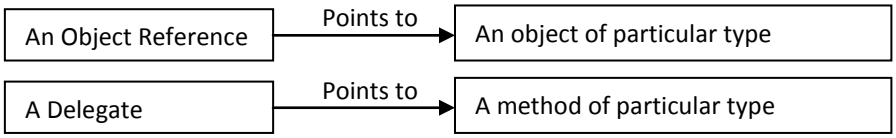
Example:

```
int[][] arr = new int[2][];
arr[0] = new int[5]{1,3,5,7,9};
arr[0] = new int[4]{2,4,6,8};
for(int i=0; i<arr.Length; i++)
{
    System.Console.WriteLine("Elements ({0}): ", i);
    for (int j=0; j< arr[i].Length; j++)
    {
        System.Console.WriteLine("{0}{1}", arr[i][j], j==(arr[if].Length-1)?"":"");
    }
```

```
    System.Console.WriteLine();
}
```

Delegates

A delegate is a type that references a method. Delegates are similar to object reference, but are used to reference methods instead of objects. The type of a delegate is the type or signature of the method.



Delegates are similar to the function pointers in C++ but are type safe. Delegates allow methods to pass as parameters.

```
using System;
delegate void SampleDelegate(string message);
class MainClass
{
   static void SampleDelegateMethod(string message)
   {
      Console.WriteLine(message);
   }
   static void Main()
   {
      //instantiation with named method:
      SampleDelegate d1 = SampleDelegateMethod;

      // note that this is just reference, method not executed.
            //Compiler executes the method if it has ()

            //Instantiate with anonymous method:
      SampleDelegate d2 = delegate(string message)
      {
         Console.WriteLine(message);
      };
      d1("Hello");
      d2("World");
   }
}
```

A delegate can be instantiated by associating it either with a named or anonymous method. For use with named method, the delegate must be instantiated with a method that has acceptable signature. For use with anonymous method, the delegate and the code to be associated with it are declared together.

```
using System;
delegate int intDelegate(int x, int y);
static void Main()
{
   myDelegate delMethod = null;
   Console.WriteLine("Enter Choice (+,-,/)");
   char choice = (char)Console.Read();
   switch(choice)
   {
      case '+';
```

```
            delMethod = new intDelegate(add);
            break;
        case '-';
            delMethod = new intDelegate(sub);
            break;
        case '/';
            delMethod = new intDelegate(div);
            break;
        default:
            Console.WriteLine("Bad Input");
            break;
    }
    int result = delMethod(10,5);
    Console.WriteLine("The Result of the Operation is: "+result);

    Static int add(int x, int y)
    {
        return x+y;
    }
    Static int sub(int x, int y)
    {
        return x-y;
    }
    Static int div(int x, int y)
    {
        return x/y;
    }
}
```

Event and Event Handling

An event is a mechanism for a class to provide notifications to clients of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interface: typically, the classes that represent control is in the interface have events that are notified when the user does something to the control (for example: click a button).

Events provide a useful way for objects to signal state change that may be useful to clients of the object.

An event is basically a message which is said to be fired or triggered when the respective action occurs. A class that sends (or raises) the event is called the publisher (sender) and the class that receives (or handle) the event are called subscribers (consumer) and the method which is used to handle a particular event is called event handler.

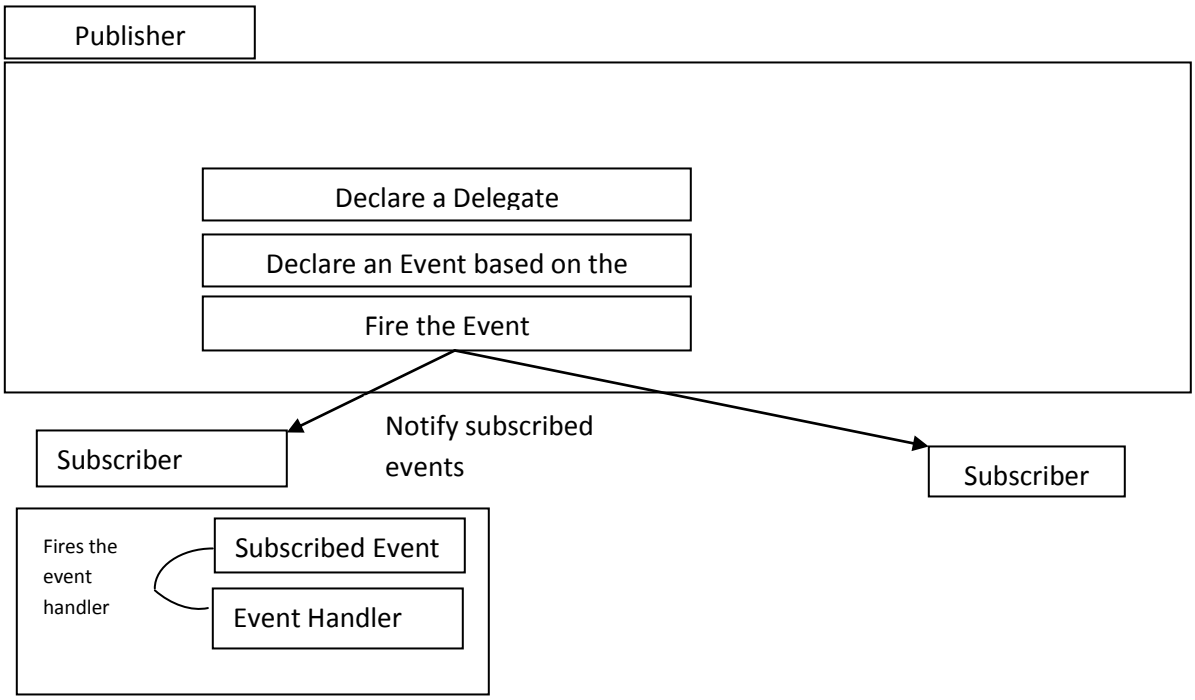
- Publisher determines when an event is raised; the subscriber determines what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple subscribers.
- Events that have no subscribers are never raised.

Multicast Delegate

It is a delegate which holds the reference of more than one method. Multicast delegates must contain only methods that return *void*.

```
delegate void Delegate_Multicast(int x, int y);
class DelegateTestClass
```

```
{
   static void Method1(int a, int b)
   {
      System.Console.WriteLine("method1");
   }
   static void Method2(int a, int b)
   {
      System.Console.WriteLine("method2");
   }
   public static void Main()
   {
      Delegate_Multicast func = new Delegate_Multicast(Method1);
      func + = new Delegate_Multicast(Method2);
      func(1,2); //method11 and method2  are called
      func - = new Delegate_Multicast(Method1);
      func(2,3); //only method2 is called
   }
}
```



Event Handling

Events are implemented as multicast delegates. Events are defined using the event keyword.

Steps for event implementation and handling are:-

1. Declare an event:

To declare an event inside a class, first a delegate type of the event must be declared, if none is already declared.

public delegate void ChangedEventHandler(object sender, Eventargs e);

The delegate type defines the set of arguments that are passed to the method that handles the event.

Next, declare the event itself.

public event ChangedEventHandler Changed;

An event is declared like a field of delegate type, except that the keyword precedes the event declaration, following the modifiers. Events usually are declared public, but any accessibility modifiers are allowed.

2. Invoking an event:

Once a class has declared an event, it can treat that event just like a field of the indicated delegate type. The field will either be null, if no client has hooked up a delegate to the event, or else it refers to a delegate that should be called when the event is invoked. This invoking an event is generally done by first checking for null and then calling the event.

If(Changed != null)
Changed(this,e);

Invoking an event can only be done from within the class that declares the event.

3. Hooking up to an event:

From outside the class that declared it an event looks like a field, but access to that field is very restricted. The only things that can be done are:

- compose a new delegate onto that field
- remove a delegate from a (possible composite)field

This is done with the += and - = operators. To begin receive event notifications; client code first creates a delegate of the event type that refers to the method that should be invoked from the event. Then it composes that onto any other delegates that the event might be connected to using +=.

List.Changed += new ChangedEventHandler(ListChanged);

When the client code is done receiving event notifications, it removes its delegate from the event by using operator - =

Example:

```
using System;
namespace MyCollections
{
   using System.Collections;
   public delegate void ChangedEventHandler(object sender, EventArgs e);
   public class ListWithChangedEvent ArrayList
   {
      public event ChangedEventHandler Changed;
      void OnChanged(EventArgs e)
      {
         if(Changed != null)
            Changed(this,e);
      }
      public override int Add(object Value)
      {
         int i = base.Add(Value);
         OnChanged(EventArgs.Empty);
         return i;
      }
      public override void Clear()
      {
```

```
            base.clear()
                OnChanged(EventArgs.Empty);
        }

    }
}
namespace TestEvents
{
    using MyCollections;
    class EventListner
    {
        private ListWithChangedEvent List;
        public EventListner(ListWithChangedEvent list)
        List=list;
        //Add "ListChanged" to the changed event on "List"
        list.Changed += new ChangedEventHandler(listChanged);
        private void listChanged(object sender, EventArgs e)
        {
            Console.WriteLine("This is called when the event fires.");
        }
        public void Detach()
        {
            List.Changed -= new ChangedEventHandler(listChanged);
            list = null;
        }
    }
    class Test
    {
        public static void Main()
        {
            ListWithChangedEvent list = new ListWithChangedEvent();
            EventListner listener = new EventListner(list);
            list.Add("item i");
            list.Clear();
            listener.Detach();
        }
    }
}
```

Indexers

Indexers are a syntactic convenience that enable you to create a class, struct, or interface that client applications can access just as an array. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class named TempRecord that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array named "temps" of type float to represent the temperatures, and a DateTime that represents the date the temperatures were recorded. By implementing an indexer in this class, clients can access the temperatures in a TempRecord instance as float temp = tr[4] instead of as float temp = tr.temps[4]. The indexer notation not only simplifies the syntax for client applications; it also makes the class and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the this keyword, as in this example:

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

The following example shows how to declare a private array field, temps, and an indexer. The indexer enables direct access to the instance tempRecord[i]. The alternative to using the indexer is to declare the array as a public member and access its members, tempRecord.temps[i], directly.

Notice that when an indexer's access is evaluated, for example, in a **Console.Write** statement, the get accessor is invoked. Therefore, if no **get** accessor exists, a compile-time error occurs.

```
class TempRecord
{
   // Array of temperature values
   private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
                                   61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

   // To enable client code to validate input
   // when accessing your indexer.
   public int Length
   {
      get { return temps.Length; }
   }
   // Indexer declaration.
   // If index is out of range, the temps array will throw the exception.
   public float this[int index]
   {
      get
      {
         return temps[index];
      }

      set
      {
         temps[index] = value;
      }
   }
}

class MainClass
{
   static void Main()
   {
      TempRecord tempRecord = new TempRecord();
      // Use the indexer's set accessor
      tempRecord[3] = 58.3F;
      tempRecord[5] = 60.1F;

      // Use the indexer's get accessor
      for (int i = 0; i < 10; i++)
      {
         System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);
      }

      // Keep the console window open in debug mode.
      System.Console.WriteLine("Press any key to exit.");
      System.Console.ReadKey();

   }
```

```
}
/* Output:
      Element #0 = 56.2
      Element #1 = 56.7
      Element #2 = 56.5
      Element #3 = 58.3
      Element #4 = 58.8
      Element #5 = 60.1
      Element #6 = 65.9
      Element #7 = 62.1
      Element #8 = 59.2
      Element #9 = 57.5
   */
```