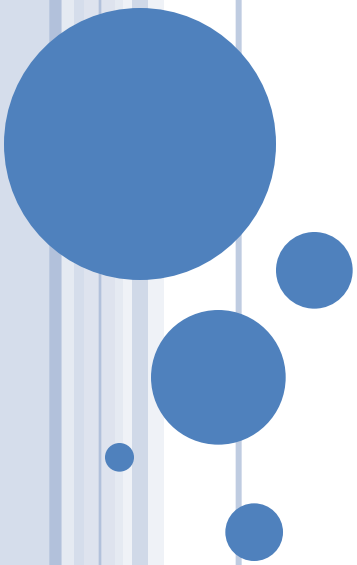


Error Handling and Tracing

Introduction to Exceptions

- Exception is an error object.
- Whenever something exceptional happens, exception is *thrown*.
- Exceptions are *caught* to be handled i.e. error handling.

  e.g. Writing to a file which is deleted.

Structured Exception Handling

- A way to organize blocks of code in a structure that will handle errors.

- In C#, structured exception handling is incorporated using *try…catch…finally* block. In VB.NET, *Try…Catch…Finally…End Try*.

Structured Exception Handling

- *try* Block has the code that might throw an exception.
- *catch* Block has the code to process the exception
- *finally* Block code is always executed, though no error occurs.

Structured Exception Handling

- *try* block can be followed by multiple *catch* blocks
- *finally* block is optional one.

System.Exception class

- It is the base class for all exceptions.

- Properties:

  - Message: The descriptive text for the exception.
  - TargetSite: Returns the name and signature of the procedure in which the exception was first thrown.
  - Source: Sets or returns the name of the component in which the exception was thrown.
  - HelpLink: URL of help file for the exception.

System.Exception Class

- *ApplicationException* and *SystemException* are two derived types of *Exception* class.
- Custom exception classes must be derived from *ApplicationException*.
- *SystemException* is for .NET CLR exception.

Common Exceptions

- DivideByZeroException
- IndexOutOfRangeException
- NullReferenceException
- InvalidCastException
- FileNotFoundException
- FormatException

```
try
{
    x=x/y;
     //Unreached code if y=0
    y=Convert.ToInt32(10^x);
}
catch(Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Multiple Exceptions

- If *try* block code is capable of throwing multiple exceptions, it must be followed by multiple catch blocks.

- Always code *catch* blocks for more specific exception first and then general one.

  i.e. *DidvideByZeroException* must be handled before *Exception*.

Exception Handling

```
try
{
    x=x/y;
    z=x+y;
}
catch(DivideByZeroException ex)
{
    …
}
catch(Exception ex)
{
    …
}
```

Custom Exception Class

```
public class CarIsDeadException: ApplicationException
{
    public CarIsDeadException(){}
    public CarIsDeadException(string msg): base(msg)
    {}
}
```

Throwing an Exception

- *throw* keyword is used with exception object.

throw new CarIsDeadException("Engine overheated");

Re-throwing an Exception

- One can use *throw* in *catch* block.

- It is refereed as re-throwing exception to the previous caller up in the call stack.

```
try
{
    …
}
catch(Exception ex)
{
    throw ex;
}
```

InnerException

" When you encounter an exception while processing another exception, one should record the new exception object as an "inner exception" within a new object of the same type as the initial exception. "

- Exception constructor must have parameter "inner" to hold new exception.

Error Pages

- One can create the rich information pages that are shown to describe unhandled errors.
- Custom error pages setting is in web.config file.

ASP.NET Error Page

Server Error in '/ASPNETDemo' Application.

Cannot find table 0.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the

Exception Details: System.IndexOutOfRangeException: Cannot find table 0.

Source Error:

```
Line 15:     {
Line 16:         DataSet ds=new DataSet();
Line 17:         Label1.Text=ds.Tables[0].Rows[0][0].ToString();
Line 18:     }
Line 19:
```

Source File: c:\Documents and Settings\pradip\My Documents\Visual Studio 2005\WebSites\ASPNETDemo\Default.aspx.cs **Line:** 17

Stack Trace:

```
[IndexOutOfRangeException: Cannot find table 0.]
   System.Data.DataTableCollection.get_Item(Int32 index) +107
   _Default.Page_Load(Object sender, EventArgs e) in c:\Documents and Settings\pradip\My Documents\Visual Studio 2005\WebSites\ASPNETDer
   System.Web.Util.CalliHelper.EventArgFunctionCaller(IntPtr fp, Object o, Object t, EventArgs e) +31
   System.Web.Util.CalliEventHandlerDelegateProxy.Callback(Object sender, EventArgs e) +68
   System.Web.UI.Control.OnLoad(EventArgs e) +88
   System.Web.UI.Control.LoadRecursive() +74
   System.Web.UI.Page.ProcessRequestMain(Boolean includeStagesBeforeAsyncPoint, Boolean includeStagesAfterAsyncPoint) +3035
```

ASP.NET Error Page

- The previous error page is shown only for local requests that access the ASP.NET application through the *http://localhost* domain.
- ASP.NET doesn't create a rich error page for requests from other computers, which shows generic error page.
- The generic page lacks any specific details about the type of error or the offending code.

Error Modes

- ASP.NET rich error page is shown in local domain only as the <customErrors> section of web.config file set *mode=RemoteOnly*.
- Mode can also be set to *Off* and *On*.
- RemoteOnly
  - The default setting, which uses rich ASP.NET error pages only when the developer is accessing an ASP.NET application on the current machine.

Error Modes

- Off
  - Configures rich error pages (with source code and stack traces) for all unhandled errors, regardless of the source of the request.
- On
  - ASP.NET error pages will never be shown. When an unhandled error is encountered, a corresponding custom error page will be shown if one exists.
  - Otherwise, ASP.NET will show the generic message explaining that application settings prevent the error details from being displayed and describing how to change the configuration.

Custom Error Page

- One can create a single generic error page and configure ASP.NET to use it by modifying the web.config file.

```
<system.web>
        <customErrors defaultRedirect="someerror.aspx" />
</system.web>
```

The custom error page comes into effect only if ASP.NET is handling the request.

Custom Error Page Behavior

- If ASP.NET encounters an HTTP error while serving the request, it will forward the user to the **someerror.aspx** web page.

- If ASP.NET encounters an unhandled application error and it isn't configured to display rich error pages, it will forward the user to the **someerror.aspx** web page.

Custom Error Page Behavior

- If ASP.NET encounters an unhandled application error and it is configured to display rich developer-targeted error pages, it will display the rich error page instead.

- If an error occurs in the error page itself, it will display the normal client error page with the generic message.

Specific Custom Error Pages

- One can create error pages targeted at specific types of HTTP errors such as 404 Not Found Error or Access Denied.

- To define an error-specific custom page, add an <error> element to the <customErrors> element. The <error> element identifies the HTTP error code and the redirect page.

Specific Custom Error Pages

```
<system.web>
        <customErrors defaultRedirect="someerror.aspx">
                <error statusCode=404 redirect="My404.aspx" />
        <customErrors>
</system.web>
```

The user will be redirected to the **My404.aspx** page when requesting an ASP.NET page that doesn't exist.
The custom error page comes into effect only if ASP.NET is handling the request.

Trapping Errors on Page-Level

- Each ASP.NET page is derived from *System.Web.UI.Page* class.
- The page object helps to trap page-level errors.
- For this, one need to override *OnError* method of page.

Trapping Errors on Page-Level

```
protected override void OnError(EventArgs e)
    {
        // At this point we have information about the error
        HttpContext ctx = HttpContext.Current;
        Exception exception = ctx.Server.GetLastError();
        string errorInfo =
          "<br>Offending URL: " + ctx.Request.Url.ToString() +
          "<br>Source: " + exception.Source +
          "<br>Message: " + exception.Message +
          "<br>Stack trace: " + exception.StackTrace;
        ctx.Response.Write(errorInfo);
        // -----To let the page finish running we clear the error
        // ----------------------------------------------------------------------
        ctx.Server.ClearError();

        base.OnError(e);
    }
```

Trapping Errors on Application Level

- *HttpApplication* fires *Error* event.
- One can implement his/her own error handler in *Global.asax* file.

Trapping Errors on Application Level

```
protected void Application_Error(Object sender, EventArgs e)
  {
     // At this point we have information about the error
     HttpContext ctx = HttpContext.Current;
     Exception exception = ctx.Server.GetLastError();
     string errorInfo =
        "<br>Offending URL: " + ctx.Request.Url.ToString() +
        "<br>Source: " + exception.Source +
        "<br>Message: " + exception.Message +
        "<br>Stack trace: " + exception.StackTrace;
     ctx.Response.Write(errorInfo);
     // --To let the page finish running we clear the error
     // ------------------------------------------------
     ctx.Server.ClearError();
  }
```

Tracing

- Tracing is to monitor the execution of ASP.NET Application.

- Two Types

  - Page-Level Tracing

  - Application-Level Tracing

Page-Level Tracing

- ASP.NET tracing can be enabled on a page-by-page basis by adding Trace="true" to the Page directive.

```
<%@ Page Language="C#" Trace="true" %>
```

- One can also enable tracing using the built-in *Trace* property. By default, tracing applies to local requests.

```
protected void Page_Load(Object sender, EventArgs e)
{
        Trace.IsEnabled=true;
}
```

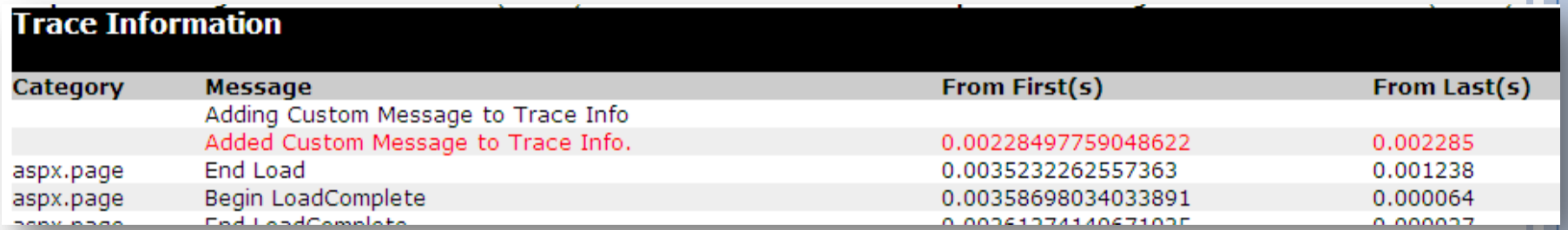
Writing Trace Information

- To write a custom trace message, you use the *Write()* method or the *Warn()* method of the Trace property.

- *Warn()* displays the message in red color.

```
Trace.Write("Adding Custom Message to Trace Info");
Trace.Warn("Added Custom Message to Trace nfo.");
```

Writing Trace Information

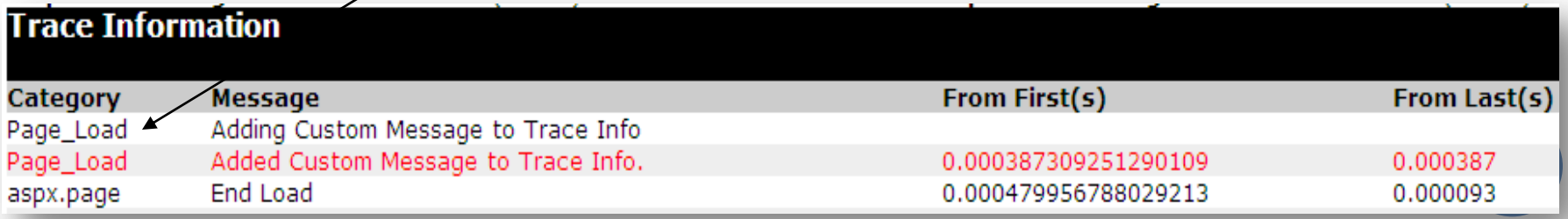


Trace Information

Category	Message	From First(s)	From Last(s)
	Adding Custom Message to Trace Info		
	Added Custom Message to Trace Info.	0.00228497759048622	0.002285
aspx.page	End Load	0.0035232262557363	0.001238
aspx.page	Begin LoadComplete	0.00358698034033891	0.000064

- An overloaded method of Write() or Warn() that allows us to specify the category.

```
Trace.Write("Page_Load","Adding Custom Message to Trace Info");
Trace.Warn("Page_Load","Added Custom Message to Trace nfo.");
```



Trace Information

Category	Message	From First(s)	From Last(s)
Page_Load	Adding Custom Message to Trace Info		
Page_Load	Added Custom Message to Trace Info.	0.000387309251290109	0.000387
aspx.page	End Load	0.000479956788029213	0.000093

Order of Trace Messages

- Trace messages are listed in the order they were written by the code.

- One can specify that messages should be sorted by category using the *TraceMode* attribute in the *Page* directive.

```
<%@ Page Language="C#" Trace="true" TraceMode="SortByCategory" %>
```

Or Programmatically

```
Trace.TraceMode=TraceMode.SortByCategory;
```

Application-Level Tracing

- Application-level tracing enables tracing for an entire application.
- Application level tracing settings are in web.config file.

```
<system.web>
        <trace enabled="true" localOnly="true" pageOutput="true“
                traceMode="SortByTime" />
</system.web>
```

Application-Level Tracing Options

- requestLimit
  - The number of HTTP requests for which tracing information will be stored.
  - When the maximum is reached, the information for the oldest request is abandoned every time a new request is received.
- pageOutput
  - Determines whether tracing information will be displayed on the page.
  - If false, view trace info requesting *trace.axd* in browser.

Application-Level Tracing Options

- traceMode
  - Determines the sort order of trace messages.
- localOnly
  - Determines whether tracing information will be shown only to local clients or can be shown on remote clients also.
- mostRecent
  - If true, ASP.NET keeps only the most recent trace messages. Abandons oldest trace info.
  - If false, ASP.NET do not store trace info when maximum requestLimit is reached.