



# Coding conventions & guidelines

Demachina  
[www.demachina.com](http://www.demachina.com)

**UNREGISTERED COPY**



## Document Control

Draft	Author	Date	Comment
1.0	-	15-01-2003	Initial version
1.1	-	21-01-2003	Added C-Sharp and PL/SQL sections
1.2	-	09-09-2003	Added Java section
1.3	-	28-09-2003	Updated Java section
1.4	-	23-03-2004	Cleanup, Company name change
1.5	-	09-03-2005	Template style change

## Purpose and audience of document

The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of an application so that you and others can easily read and understand the code.

Good coding conventions result in precise, readable, and unambiguous source code that is consistent with other language conventions and as intuitive as possible.

A general-purpose set of coding conventions should define the minimal requirements necessary to accomplish the purposes discussed above, leaving the programmer free to create the program's logic and functional flow.

The object is to make the program easy to read and understand without cramping the programmer's natural creativity with excessive constraints and arbitrary restrictions.

To this end, the conventions suggested in this document are brief and suggestive. They do not list every possible object or control, nor do they specify every type of informational comment that could be valuable. Depending on your project you may wish to extend these guidelines to include additional elements

## Summary

Writing a maintainable solution is not hard, it just takes discipline. Follow the simple guidelines described in this document and you won't encounter a single problem ever. Promise ;')

.

**THIS IS NOT A FREE DOCUMENT! PLEASE BUY THE OFFICIAL VERSION FROM [WWW.DEMACHINA.COM/PRODUCTS/SWAT/](http://WWW.DEMACHINA.COM/PRODUCTS/SWAT/). THIS WILL GIVE YOU ACCESS TO AN EDITABLE VERSION OF THE DOCUMENT.**

## Disclaimer

© Demachina. All rights reserved. No part of this document may be altered, reproduced or distributed in any form without the expressed written permission of Demachina.

This document was created strictly for information purposes. No guarantee, contractual specification or condition shall be derived from this document unless agreed to in writing. Demachina reserves the right to make changes in the products and services described in this document at any time without notice and this document does not represent a commitment on the part of Demachina in the future.

While Demachina uses reasonable efforts to ensure that the information and materials contained in this document are current and accurate, Demachina makes no representations or warranties as to the accuracy, reliability or completeness of the information, text, graphics, or other items contained in the document. Demachina expressly disclaims liability for any errors or omissions in the materials contained in the document and would welcome feedback as to any possible errors or inaccuracies contained herein.

Demachina shall not be liable for any special, indirect, incidental, or consequential damages, including without limitation, lost revenues or lost profits, which may result from the use of these materials. All offers are non-binding and without obligation unless agreed to in writing.

## Contents

1	Introduction	8
1.1	How to use this document	8
2	Coding Standards	9
2.1	General	9
2.1.1	Copyright message & page header	9
2.1.2	Document the code	10
2.1.3	Documenting functions	11
2.1.4	Has unused code been removed?	11
2.1.5	Formatting Code	12
2.1.6	Naming Conventions	13
2.1.7	Error handling	13
2.1.8	Presenting errors to the user	13
2.1.9	Declaring variables	14
2.1.10	Concatenating strings	14
2.1.11	Working with arrays	14
2.1.12	Validation & Cleaning up data	14
2.1.13	Test Performance, Scalability, and Reliability	14
2.1.14	Separate presentation from business logic	15
2.1.15	Caching	15
2.1.16	Directory Structure	15
2.1.17	Acquire Resources Late and Release Them Early	15
2.1.18	Transactions	16
2.1.19	Reading & writing configuration settings	16
2.2	VB & VBScript	17
2.2.1	Copyright message & page header	17
2.2.2	Document the code	17
2.2.3	Documenting functions	17
2.2.4	Formatting Code	17
2.2.5	Naming conventions	18
2.2.6	Encapsulate code in classes	20
2.2.7	Error handling	21
2.2.8	Declaring variables	23
2.2.9	Concatenating strings	23
2.2.10	Including files	24
2.2.11	Clean up your objects	24
2.2.12	Transactions (ADO)	24
2.2.13	Reading & writing configuration settings	25
2.3	C#	25
2.3.1	Copyright message & page header	25
2.3.2	Document the code	26
2.3.3	Documenting functions	27
2.3.4	Formatting Code	27
2.3.5	Naming conventions	27
2.3.6	Object Orientation	28
2.3.7	Error handling	28
2.3.8	Concatenating strings	29
2.3.9	Call <i>Dispose()</i> when required	30
2.3.10	Caching	30
2.3.11	Transactions (ADO.NET)	30
2.4	JavaScript	31
2.4.1	Copyright message & page header	31
2.4.2	Document the code	31

2.4.3	Documenting functions	32
2.4.4	Formatting Code	32
2.4.5	Naming conventions	33
2.4.6	Declaring variables	33
2.4.7	Error handling	33
2.4.8	Validating forms	34
2.4.9	Concatenating strings	34
2.4.10	Including files	35
2.4.11	Clean up your objects	35
2.4.12	Reading & writing configuration settings	35
2.5	ASP	35
2.5.1	Copyright message & page header	35
2.5.2	Document the code	36
2.5.3	Documenting functions	36
2.5.4	Formatting Code	36
2.5.5	Error handling	37
2.5.6	Presenting errors to the user	37
2.5.7	Separate presentation from business logic.	38
2.5.8	Validate input parameters	38
2.5.9	Including files	38
2.5.10	Use a proper default document	39
2.5.11	Transactions	39
2.5.12	Are HTTP caching headers being used?	39
2.5.13	Are global variables defined in the global.asa?	40
2.5.14	Reading & writing configuration settings	40
2.6	ASP.NET	40
2.6.1	Copyright message & page header	40
2.6.2	Separate presentation from business logic	41
2.7	Microsoft T-SQL	42
2.7.1	Copyright message & page header	42
2.7.2	Documenting stored procedures & functions	42
2.7.3	Formatting Code	42
2.7.4	Naming conventions	43
2.7.5	Use Named arguments in complex SP calls	46
2.7.6	Minimizing Code	46
2.7.7	Error handling	46
2.7.8	Transactions	47
2.7.9	Take care of locking	48
2.7.10	Reading & writing configuration settings	48
2.7.11	Pre process data	49
2.7.12	Miscellaneous	49
2.8	Oracle PL/SQL	51
2.8.1	Copyright message & page header	51
2.8.2	Documenting stored procedures & functions	52
2.8.3	Formatting Code	52
2.8.4	Naming conventions	53
2.8.5	Use Named arguments in complex SP calls	55
2.8.6	Minimizing Code	55
2.8.7	Error handling	56
2.8.8	Transactions	56
2.8.9	Take care of locking	56
2.8.10	Pre process data	56
2.8.11	Use packages in stead of stand alone stored procedures	57
2.8.12	Use data dictionary types when declaring variables	58
2.8.13	Miscellaneous	59
2.9	HTML	60
2.9.1	Copyright message & page header	60
2.9.2	Formatting Code	60

2.9.3	Directory structure	61
2.9.4	Use style sheets	61
2.9.5	Form elements	62
2.9.6	Validate forms using JavaScript	62
2.9.7	Browser compatibility	63
2.10	Java	63
2.10.1	Copyright message & page header	63
2.10.2	Document the code	64
2.10.3	Documenting methods	64
2.10.4	Documenting the package	65
2.10.5	Formatting Code	65
2.10.6	Naming conventions	65
2.10.7	Object Orientation	66
2.10.8	Error handling	66
2.10.9	Concatenating strings	67
2.10.10	Use of Reflection	67
2.10.11	Do override equals, toString and hashCode	67
2.10.12	Logging	68
2.10.13	Design & Patterns	68
2.10.14	Unit Testing	69
2.10.15	Builds	69
2.10.16	Reading & writing configuration settings	69

# 1 Introduction

This document assumes a working experience with web based technologies, SQL, VB and a general feel of common sense. For those who lack the latter some tips are provided below. Hopefully this information is an insult to your intellect, in which case we are off to a good start.

- **Communicate:** If you don't understand something or feel like you are re-inventing the wheel, check out the company knowledge base, google or ask your colleagues. It won't make you look stupid if you do.
- **Think before you act:** Most disasters can be avoided by taking a couple of minutes to design what you are going to do. It doesn't matter if you are creating a recommendation engine or a simple directory structure, if you make fundamental mistakes at the beginning of your project the results might haunt you and your team for years.
- **Be consistent:** No explanation necessary.

This document contains a lot of information. If you don't understand something don't read over it but raise a question IMMEDIATELY as this document contains no unimportant information.

Have no fear, the sarcasm is over for the rest of the document, let's get on with the useful stuff.

## 1.1 How to use this document

Depending on your role read the following chapters:

Role	Chapters
ASP Developer	Chapter 2.1, 2.2, 2.4, 2.5, 2.7, 2.8, 2.9
ASP.NET Developer	Chapter 2.1, 2.3, 2.4, 2.6, 2.7, 2.8, 2.9
Java Developer	Chapter 2.1
MSSQL Database Developer	Chapters 2.1, 2.7
Oracle Database Developer	Chapters 2.1, 2.8
Graphics designer / HTML developer	Chapters 2.1, 2.4, 2.9
Project Manager	Chapter 2.1
Consultant / Business analyst	Chapter 2.1



## 2 Coding Standards

### 2.1 General

This general section describes standards that are true for any kind of programming that is done, independent of what language is being used. Language specific topics are discussed in their own chapters.

#### 2.1.1 Copyright message & page header

Not interesting from a readability point of view, but when looking at it from a legal and marketing perspective it makes sense to include a copyright noticed and some additional information in each file that is published, especially considering HTML and JavaScript files can potentially be viewed by customers and competitors.

Make sure that '<Solution Name>' is replaced with the proper project name such as 'Perfyt'

While we are adding a page header we might as well put some other useful information in it. Listed below is a sample page header (VBScript format) that contains the minimal information any page (or source file) should include. The language specific page headers are described in their own chapter.

```
! *****
!                                     <Solution Name>
!
!                                     Copyright 2005, Demachina
!                                     www.demachina.com
!
!                                     All rights reserved
!
!                                     Document author: Jeroen Ritmeijer
!
! FILENAME       : orderimport.vbs
! PURPOSE        : Check if any order...
! MODULE         : Order imports, For more info see: specs.doc
! DEPENDENCIES   : includes/inc_database.vbs
!
! TO-DO          : -
! KNOWN ISSUES   : -
! TAB SIZE       : 4
! HISTORY        : 04/03/2002 - JR - Created file
!                : 07/03/2002 - GT - Performed code review
! *****
```

**Filename:** Useful when you are reading a hardcopy and you want to know what the filename is.

**Purpose:** A brief description of the file and what it does.

**Module:** The name of the (sub) project this file belongs to.

**Dependencies:** What external dependencies does this file have? List such things as COM objects, registry keys, database connections and include files.

**To-do:** Any outstanding issues such as 'add error handling'.

**Known Issues:** Known issues that are not severe enough to stop shipping the code. For example *'Might not work after 2009, but we can safely assume this code will no longer be used by then'*.

**Tab size:** The tab size that has been used to create and edit this file. This is to prevent people from constantly reformatting the file to their favourite tab size.

**History:** The history contains an overview of the major changes (usually complete revisions or bug fixes) that have been made to the file. Don't fill this in on a day by day basis or every time you add one line of code as part of the development process. In theory this history should be the same as the one in your source code control system, however that system is not always at hand when you look at the file (because you are looking at a printout or you are off site).

### 2.1.2 Document the code

Although it sounds obvious, the reality is that many thousands of lines of code are written without a single comment in it. The reason for this is usually external pressure or job security. However when you are working in a team or if you have to look back at your own code after a year, the benefit of having comments in the code is tremendous.

Comment the following

- **Page header:** See 2.1.1
- **Individual functions:** See 2.1.3
- **Local and global variables:** Add a small comment just before or on the same line the variable is defined on.
- **Code sections:** Add a single line for each logical unit of code, e.g. 'Open database connection and Initialize recordsets' and 'In case of a failure write to the event log'.

Although ideally you should write comments while (or even before) you start programming, it might not always be achievable to do this because you are working on a prototype and you are not sure yet what the final solution will look like. Therefore the general rule is that *a file needs to be documented before it is checked into the source code control system*.

Listed below is an example of a properly documented piece of code.

```
...
''' Get the files (note that this returns ALL files in the folder,
''' not just XML files)
set oFiles = GetImportFiles

''' Iterate through the files collection
if not (oFiles is nothing) then
    for each oFile in oFiles
        ''' Only process xml files
        if lcase(oFileSystem.GetExtensionName(oFile)) = "xml" then
...

```

*See, this is not exactly rocket science*

For more information see *Using comments effectively* at <http://msdn.microsoft.com/library/office/dev/odeopg/deconusingcommentseffectively.htm> and *Coding*

*techniques and programming practices* at <http://msdn.microsoft.com/library/techart/cfr.htm>.

### 2.1.3 Documenting functions

Each function needs to be documented. This does not only make it easier for your fellow programmers to read, but also allows the company to easily generate an API reference.

Independent of what language you are using a function's header should at least contain the following information:

- **Function:** The name of the function.
- **Purpose:** What the routine does, not how it does it.
- **Assumes:** List of each non-obvious external variable, control, open file, and so on.
- **Affects:** List of each affected external variable, control, file, and so on and the affect it has on it (only if this is not obvious).
- **Parameters:** Each parameter and its type (if applicable) on a separate line with in-line comments.
- **Returns:** Explanation of the returned value.

Displayed below is an example of a properly documented function header, in this case for VBScript.

```
.....  
' FUNCTION : SaveCollection  
' PURPOSE : Save a collection to the database  
' ASSUMES : An open database connection named oConn  
' AFFECTS : -  
' PARAMETERS: nCustomerID - The customer's ID.  
'             nCategoryID - The active category ID.  
' RETURNS: true or false, indicating a successful save.  
.....  
function SaveCollection(nCustomerID, nCategoryID)  
...
```

### 2.1.4 Has unused code been removed?

Before releasing your final code remove all disabled (commented out) code and other parts that are not used like functions that are no longer referenced.

Listed below is an example on how **not to do it**:

```
...  
    set oRS = createobject("adodb.recordset")  
    *** The parameters for the stored procedure have changed  
    ' oConnAdmin.SelectCurrentOrderExportRequests oRS  
    oConnAdmin.SelectCurrentOrderExportRequests nOrderID, oRS  
    set oRS.ActiveConnection = nothing  
    *** Did an error occur?  
    if err then  
...  
...
```

In the example above the programmer has tried to indicate what has changed in the code. However, if we want to know what has changed we can either look at the page header (See 2.1.1) or perform a *diff* in the source code control system.

### 2.1.5 Formatting Code

Although not that interesting from a customer perspective, proper code formatting is very important for the team maintaining the solution, which may be completely different from the team that created the solution in the first place.

Just follow the simple rules outlined below:

- **Indent your code:** Indented code is much easier to read and follow.
- **Break up long statements:** Long statements are extremely difficult to read when everything is contained on the same line. If the language permits it distribute the statement over multiple lines
- **Space your code:** Don't leave empty lines all over the place, a single line between code fragments and two lines between function definitions will do for most cases.
- **Watch you tab size:** Try to use your development environment's default tab size as this is most likely the tab size your fellow coders are using. Try not to mix tabs and spaces to indent your code.

Listed below is an example of a properly indented VBScript function:

```
function UpdateOrderStatus(oRSOrderHeaders)
    dim nStatus, bSuccess

    bSuccess = true

    '** Depending on the existing status code, update it to a new code
    select case oRSOrderHeaders("Status")
        case 1      '** New orders
            nStatus = 2
        case 2031   '** The order has been cancelled
            nStatus = 203
        case 2032   '** The delivery date has changed
            nStatus = 2
        case 2033   '** The delivery address has changed
            nStatus = 2
        case else
            LogMessage "Unrecognised Status code " & _
                cstr(oRSOrderHeaders("Status")) & _
                "for order " & cstr(oRSOrderHeaders("OrderID"))
            bSuccess = false
    end select

    '** Is everything still OK? Then update the order status
    if bSuccess then
        LogMessage "updating order status for " & _
            cstr(oRSOrderHeaders("OrderID")) & " to " & cstr(nStatus)
        on error resume next
        oConnWeb.SMUpdateOrderStatus oRSOrderHeaders("OrderID"), nStatus
        on error goto 0
        '** Did an error occur?
        if Err then
            LogMessage "An error occurred while updating the order " & _
                "status for OrderID " & cstr(oRSOrderHeaders("OrderID")) & _
                ". The error is: " & err.description
            bSuccess = false
        else
    
```

```

        bSuccess = true
    end if
end if

UpdateOrderStatus = bSuccess
end function

```

### 2.1.6 Naming Conventions

Variables, controls, and procedures should be named clearly enough that inline commenting is needed only for complex implementation details.

Don't use variable names such as 'a', 'qc' or whatever unclear abbreviation you can think off. However loop variables are allowed to have such names as 'i' and 'j' for historical reasons.

Unfortunately every language specification recommends different notations for functions and variables. We try to stay as close to the original recommendations as possible.

This document frequently refers to the following capitalisation styles:

Pascal case	The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters.	BackColor
Camel case	The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.	backColor
Uppercase	All letters in the identifier are capitalized.	System.IO

More information can be found in the language specific chapters.

### 2.1.7 Error handling

Error handling needs to be put in place, period! Don't assume everything will always work just because it worked the last time you tested your code. There are a multitude of external and internal factors that can influence the health of your program. These can range from database connections disappearing to running out of bounds in an array.

Error handling is extremely language specific and is therefore discussed in more detail in the next few chapters.

### 2.1.8 Presenting errors to the user

Having error handling by itself is not good enough, you should also have a strategy on how to display errors to the user and how to notify the administrative staff that an unexpected system error has occurred.

The error that should be displayed to the user should be non-technical. It is of no importance to the user that there is a problem with the database because the ADO parameter being passed is of the wrong type. However this information is very interesting to the administrator.

Because displaying errors is a presentation layer task, it will be discussed in more detail in the chapter about ASP (paragraph 2.5.5 to be more specific).

#### **2.1.9 Declaring variables**

A number of modern programming languages, specifically scripting languages, allow you to skip the declaration of parameters. As you should know, this is extremely error prone and should be avoided in all cases.

When declaring variables, always define them with the smallest possible scope, don't make everything global, this is extremely bad practice.

#### **2.1.10 Concatenating strings**

Keep in mind that concatenating many strings in a loop can become very slow due to constant memory (re)allocation requests. Either use a chunking technique or a smart string formatter to create large strings. Language dependent implementations are discussed in the applicable chapters.

#### **2.1.11 Working with arrays**

Our convention is that arrays start at 0, independent of what language is being used.

#### **2.1.12 Validation & Cleaning up data**

Naturally data, especially when it has been entered by a user, needs to be validated. To prevent every layer of software to implement its own validation it is important to pick a validation strategy and validate in one place only. This validation should ideally take place as close to the *source* as possible (in the GUI) to make sure invalid data is not passed through all the layers of the system before it is bounced back. This also makes error handling a lot easier.

On the other end of the spectrum, data needs to be cleaned up before it is passed to the backend, for example to remove characters that are not allowed by the full text indexer of a database. This cleaning up should take place as close to the *destination* as possible to maximize the reuse of logic. If we would perform this cleaning up in the User interface then we would need to re-implement it if we would add another channel. This rule, however should not conflict with the statements made in 2.7.6 (Minimise the use of T-SQL code). Therefore, the ideal place to put this 'clean up logic' is in a custom business object, just before the data is passed to the data layer.

#### **2.1.13 Test Performance, Scalability, and Reliability**

So that web page you just developed seems pretty snappy if you hit refresh a couple of times in your browser. This, however, does not properly reflect real life circumstances where dozens, if not hundreds, of customers are hitting the same part of the database.

Therefore a browser should only be used to perform functionality and usability tests, all other testing should be performed by load simulation software. Initial

tests should be performed by the developer, more complete tests will be performed in a separate phase by the test department.

Although requests per second by itself doesn't mean much you should at least expect to handle 30 requests per second on a typical development machine for ASP pages. Unless you are doing something extremely complicated you should re-engineer your solution if you are getting anything less than the before mentioned figure.

In general it is better to optimise a server based application for scalability rather than for non-scalable performance.

#### **2.1.14 Separate presentation from business logic**

In order to re-use business logic when a new channel is added or when the presentation layer changes, it is best to put all logic, other than presentation specific logic, inside Business objects.

#### **2.1.15 Caching**

The solutions our company develops are usually accessed by a large number of concurrent users. Therefore it is very important to increase scalability by reducing bottlenecks. This is usually done by introducing caching mechanisms. These, however, are technology specific and therefore discussed in more detail in the chapters below.

#### **2.1.16 Directory Structure**

Before you start doing anything, think about what you are going to develop. Don't throw everything in one big directory, being it source code files or web images, but rather create a proper directory structure first.

More information on this subject can be found in 2.9.3.

#### **2.1.17 Acquire Resources Late and Release Them Early**

A common problem for developers is transitioning from the desktop to the server. Many developers coming from a desktop background have not had to worry about server issues and resource sharing. In traditional desktop applications, connecting to the server was a time-consuming process. To improve the user experience, a common approach was to acquire resources early and release them late. For example, many applications would hold an open connection to the database for the application lifetime.

This approach worked fine in traditional desktop applications, because the user base was fairly well known and controlled, and the back end was tightly coupled to the front end. However, for today's Web applications, this approach is not feasible, because of limited server resources facing increasing user bases. To scale your application across users, you need to obtain resources late, and free them as early as possible.

Pooling has helped make this approach more effective. Through pooling, multiple users can share resources with minimal wait time and minimal impact to the server. For example, when working with databases, ODBC connection



pooling and OLEDB resource pooling enable connections to be plucked from a pool to minimize database connection overhead.

This is not only true for ODBC connections, but for any resource that is either in limited supply or take a long time to instantiate. Other examples this applies to are COM objects and database transactions.

An example of a COM object / Database connection that adheres to the 'acquire late, release early' method is given below.

```
function GetExportRequests()  
    dim oRS, oConn  
  
    '** This stored procedure is part of the AdminInterface database  
    on error resume next  
        set oConn = createobject("adodb.connection")  
        set oRS = createobject("adodb.recordset")  
        '** Instantiate connection as late as possible  
        oConn.Open Application("DSN")  
        oConnAdmin.SelectCurrentOrderExportRequests oRS  
        '** Release the database connection  
        set oRS.ActiveConnection = nothing  
        oConn.close  
        set oConn = nothing  
        '** Did an error occur?  
        if err then  
            LogMessage "An error occurred while retrieving the order" & _  
                " export requests. The error is: " & err.description  
            set GetExportRequests = nothing  
        else  
            set GetExportRequests = oRS  
        end if  
    on error goto 0  
  
end function
```

### 2.1.18 Transactions

Transactions are language / technology dependent and therefore described in more detail in the language specific chapters. A couple of generic guidelines:

1. In line with 2.1.17, begin them as late as possible and commit them as early as possible.
2. Only use them when absolutely necessary. It does not make sense to wrap a single insert or update statement in a transaction.
3. Keep transactions as small as possible. When exporting orders from one machine to another, wrap every orderheader / orderline import combination in a separate transaction to make sure the entire process isn't rolled back when order 4999 out of 5000 fails. You should be thankful for the orders that made it through correctly.

### 2.1.19 Reading & writing configuration settings

To simplify deployment, ideally all settings should be specified in one single repository. For most languages the convention is to store it in the registry unless there is a real good reason to store it elsewhere. All modern languages can read from the registry, including SQL Server. Details are provided in the chapters below.



## 2.2 VB & VBScript

This chapter outlines the conventions for VB & VBScript (stand alone VBScript files). Additional recommendations, for when VBScript is used as the programming language in ASP, are made in 2.5.

### 2.2.1 Copyright message & page header

The page header is almost identical to the one described in 2.1.1 with the following exceptions:

**Command line parameters:** Stand alone VBScript files accept command line parameters. These parameters should be documented.

```
*****
'
'               <Solution Name>
'
'               Copyright 2005, Demachina
'               www.demachina.com
'
'               All rights reserved
'
'               Document author: Jeroen Ritmeijer
'
' FILENAME      : importorders.vbs
' PURPOSE       : Check if any order...
' MODULE        : Order imports, For more info see: specs.doc
' DEPENDENCIES  : includes/inc_database.vbs
'
' Command line parameters
' StoreID       : The ID of the store for which to process the order.
'
' TO-DO         : -
' KNOWN ISSUES  : -
' TAB SIZE      : 4
' HISTORY       : 04/03/2002 - JR - Created file
'               : 07/03/2002 - GT - Performed code review
*****
```

### 2.2.2 Document the code

As described in 2.1.2 with the following addition:

Comments cannot follow a line-continuation character on the same line.

### 2.2.3 Documenting functions

As described in 2.1.3

### 2.2.4 Formatting Code

As described in 2.1.5 with the following additions:

**Breaking up lengthy statements:** Use the line-continuation character, which is a space followed by an underscore ( \_), at the point at which you want the line to break. In the following example, the statement is broken into three lines:

```
...
if Err then
```

```

    LogMessage "An error occurred while updating the order " & _
               "status for OrderID " & cstr(oRSOrderHeaders("OrderID")) & _
               ". The error is: " & err.description
    bSuccess = false
...

```

Breaking up long lines makes your code easier to read, both on screen and when printed.

Please note that you cannot follow a line-continuation character with a comment on the same line.

Some limitations exist as to where the line-continuation character can be used, such as in the middle of an argument name. You can break up an argument list with the line-continuation character, but the individual names of the arguments must remain intact. Additionally you cannot have multi line string definitions. Close the string, add an '&' followed by the line-continuation character.

**Indenting error handling:** Although there is no requirement from a language point of view to do so, it is recommended to indent code that is contained in an 'on error resume next' block. This makes it look a little bit like the more elegant 'try .. catch' statements that are supported by other programming languages. For an example see the code fragment below.

```

...
on error resume next
    set oRS = createobject("adodb.recordset")
    oConnAdmin.SelectCurrentOrderExportRequests oRS
    set oRS.ActiveConnection = nothing
    '** Did an error occur?
    if err then
        LogMessage "An error occurred while retrieving the order export " & _
                   "requests. The error is: " & err.description
        set GetExportRequests = nothing
    else
        set GetExportRequests = oRS
    end if
on error goto 0
...

```

## 2.2.5 Naming conventions

The VB naming convention is very simple, prefix all variables by its type as described in the following table.

Type	Prefix
Object	o
Boolean	b
String	s
Arrays	arr
Numbers (integers, longs)	n
Dates	dt
Doubles and floats	f
Constants	capitalize variable

Do **NOT** use the ancient *Microsoft Basic* naming convention by suffixing your variable names with \$, # or % to indicate the type.

In addition to this naming convention use the following variable names for frequently used COM objects:

Type	Name
ADO Connection	oConn
ADO Recordset	oRS
ADO Command	oCMD

Additional naming guidelines:

- Begin each separate word in a name with a capital letter, as in 'FindLastRecord' and 'RedrawMyForm'.
- Begin function and method names with a verb, as in 'InitNameArray' or 'CloseDialog'.
- Begin class and property names with a noun, as in 'EmployeeName' or 'CarAccessory'.
- Begin interface names with the prefix "I", followed by a noun or a noun phrase, like 'IComponent', or with an adjective describing the interface's behavior, like 'IPersistable'. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.
- Begin event handler names with a noun describing the type of event followed by the 'EventHandler' suffix, as in 'MouseEventHandler'.
- If an event has a concept of 'before' or 'after', use a prefix in present or past tense, as in 'ControlAdd' or 'ControlAdded'.
- For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" will lead to confusion.

Finally, VB form elements are also considered to be variables and should therefore be given logical names, not the 'Command1' default name that is assigned when you place a button on a form.

Listed below are Microsoft's recommended naming conventions for form elements.

Prefix	Object type	Example
ani	Animation button	aniMailBox
bed	Pen Bedit	bedFirstName
cbo	Combo box and drop down list box	cboEnglish
chk	Checkbox	chkReadOnly
clp	Picture clip	clpToolbar
cmd (3d)	Command button (3D)	cmdOk (cmd3dOk)
com	Communications	comFax
ctr	Control (when specific type unknown)	ctrCurrent
dat	Data control	datBiblio
dir	Directory list box	dirSource

dlg	Common dialog control	dlgFileOpen
drv	Drive list box	drvTarget
fil	File list box	filSource
frm	Form	frmEntry
fra (3d)	Frame (3d)	fraStyle (fra3dStyle)
gau	Gauge	gauStatus
gpb	Group push button	gpbChannel
gra	Graph	graRevenue
grd	Grid	grdPrices
hed	Pen Hedit	hedSignature
hsb	Horizontal scroll bar	hsbVolume
img	Image	imgIcon
ink	Pen Ink	inkMap
key	Keyboard key status	keyCaps
lbl	Label	lblHelpMessage
lin	Line	linVertical
lst	List box	lstPolicyCodes
mdi	MDI child form	mdiNote
mpm	MAPI message	mpmSentMessage
mps	MAPI session	mpsSession
mci	MCI	mciVideo
mnu	Menu	mnuFileOpen
opt (3d)	Option Button (3d)	optRed opt3dRed)
ole	OLE control	oleWorksheet
out	Outline control	outOrgChart
pic	Picture	picVGA
pnl3d	3d Panel	pnl3d
rpt	Report control	rptQtr1Earnings
shp	Shape controls	shpCircle
spn	Spin control	spnPages
txt	Text Box	txtLastName
tmr	Timer	tmrAlarm
vsb	Vertical scroll bar	vsbRate

## 2.2.6 Encapsulate code in classes

Believe it or not, but even VBScript supports classes as the sample below shows.

```
class CStringClass
    dim sValue

    Private Sub Class_Initialize ' Setup Initialize event.
        Response.Write "Class Initialised<br>"
        sValue="Uninitialized"
    End Sub

    Private Sub Class_Terminate ' Setup Terminate event.
        Response.Write "Class terminating<br>"
        sValue=""
    End Sub

    function Assign( s )
        sValue=s
    end function

    function Render
```

```
        Response.Write sValue & "<br>"
    end function
end class

dim s
set s = new CStringClass

s.Assign( "Hello World" )
s.Render()
set s = nothing
```

You should however realise that the moment you start encapsulating logic inside VBScript classes you might be better off writing a COM component as this allows you to share the logic between applications more easily. Remember, separate presentation from logic (see 2.1.14 and 2.5.7).

## 2.2.7 Error handling

### 2.2.7.1 VBScript

VBScript supports a very basic type of error handling when compared to other, more modern languages. However this does not mean that you shouldn't make use of it.

Error handling can be enabled using the following line:

```
On error resume next
```

It is considered bad practice to put one of these statements in the body of your code and automatically enable error handling (or ignoring) for all code in the entire page. The recommendation is to put error handling in those places that really need it. Usually this is in the body of a function as shown in the example below.

```
Function someFunction()
Dim blah1, blah2
    '** enable error handling
    On error resume next
        '** do something that might cause an error
        ..
        ..
        '** did an error occur?
        If err then
            '** based on err.number we might decide
            '** to undertake a different action.
            '** in this case we ignore error '321'
            If err.number <> 321 then
                '** Call a generic Error handler
                HandleError Err, 1
            End if
        End if
        '** Disable error handling and continue your work
    On error goto 0
    ...
End function
```

Although there is no requirement from a language point of view to do so, it is recommended to indent code that is contained in an 'on error resume next' block. This makes it look a little bit like the more elegant 'try .. catch' statements, which are supported by other programming languages.

Error handling can be disabled using the following line:

```
On error goto 0
```

However when global error handling has been enabled, it cannot be disabled inside a function.

Make sure the Err object is tested before 'on error goto 0' is called as this statement clears the err object.

#### 2.2.7.2 Visual Basic

Visual Basic (The compiled version, not VBScript) has slightly better error handling, it can use the VBScript kind of inline error handling described above in combination with a dedicated error handling 'sub routine' using the 'on error goto' statement.

In Visual Basic, it is important to use the "On Error" statement in every significant subroutine (and function). This will allow you to handle potentially fatal errors without causing your application to crash. An example of a fatal error is trying to divide by zero.

When a subroutine, which is called by another subroutine that does not have error handling, causes an error, control reverts to the error handling on the calling routine. For this reason you can leave error handling out of the simpler low level subroutines. The following is sample code that can be the basis of your subroutines.

```
Function CalculateSomethingInteresting() As Long
    Dim nResult As Long

    On Error GoTo Err_CalculateSomethingInteresting

    '** This function might cause an error
    nResult = CalculateSomething()

Exit_CalculateSomethingInteresting:
    '** clean up objects such as database connections
    ...
    '** Return the result value
    CalculateSomethingInteresting = nResult
    Exit Function

Err_CalculateSomethingInteresting:
    '** Set the default return value
    nResult = -1
    If Err.Number = 1 Then
        '** Call some generic error logging mechanism
        HandleError Err, "some specific error description"
        Resume Exit_CalculateSomethingInteresting
    ElseIf Err.Number = 16 Then
        '** Ignore these kind of errors
        Resume Next
    Else
        '** Call some generic error logging mechanism
        HandleError Err, "some generic error description"
        Resume Exit_CalculateSomethingInteresting
    End If

End Function

'** no error handling in this function. Any errors will
'** automatically be passed to the calling function
Function CalculateSomething() As Long
    CalculateSomething = 1 / 0
End Function
```

### 2.2.8 Declaring variables

By default neither VB nor VBScript force you to declare variables. This is extremely bad practice. Use the 'option explicit' statement to automatically throw an error when an undeclared variable is encountered.

### 2.2.9 Concatenating strings

#### 2.2.9.1 Performance

Keep in mind that concatenating many strings in a loop can become very slow due to constant memory re-allocation requests. Either use a chunking technique or a smart string formatter to create large strings.

Chunking is a relatively simple solution where you perform a number of concatenations to a relatively small string. When you are finished compiling this small string (HTML for a single product) you add it to the larger string which might contain the entire product list. An example is given below.

```
Dim sProducts, sProduct, i
'** Iterate through a 1000 products
For i = 1 to 1000
    '** Formatting a single product
    sProduct = "some html" & "some html" & "some html" & "some html"
    sProduct = sProduct & "some html" & "some html" & "some html" & "some html"
    sProduct = sProduct & "some html" & "some html" & "some html"
    '** And add it to the products list
    sProducts = sProducts & sProduct
next
```

An example of a fast VB string formatter is displayed below. It should be relatively simple to port this function to VBScript.

```
Private Const INCREMENT = 50000
Private lOffset As Long

Sub AddString(ByRef Result As String, Source As String)
    Dim lLen As Long
    lLen = Len(Source)
    If (lOffset + lLen) >= Len(Result) Then
        If lLen > INCREMENT Then
            Result = Result & Space(lLen)
        Else
            Result = Result & Space(INCREMENT)
        End If
    End If
    Mid(Result, lOffset + 1, lLen) = Source
    lOffset = lOffset + lLen
End Sub
```

Tests indicate that this can be hundreds of times faster than using the normal VB string concatenation functionality. Don't go overboard with this, only use it in places where it makes sense.

#### 2.2.9.2 & and + Operators

Always use the & operator when concatenating strings. Using the + operator to concatenate may cause problems when operating on two variants. For example:

```
vntVar1 = "10.01"
```

```
vntVar2 = 11
```

```
vntResult = vntVar1 + vntVar2 'vntResult = 21.01
```

```
vntResult = vntVar1 & vntVar2 'vntResult = 10.0111
```

### 2.2.10 Including files

Including files in normal VB is simple, just add the file to your project. VBScript however is a completely different story.

In VBScript you have to create an XML based .wsf file. This format allows you to link in external files using XML elements. An example is given below.

```
<?xml version="1.0"?>
<package>
  <job>
    <!-- include the following file -->
    <script language="VBScript" src="test.vbs" />
    <script language="VBScript">
      option explicit
      '
      'main script here...
      '
    </script>
  </job>
</package>
```

More information on .wsf XML elements can be found on <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/wsoriXMLElements.asp>.

**Important:** You cannot use the '&' operator in .wsf files to concatenate strings as this is a reserved XML character. Use the '+' operator in combination with explicit casting using `cstr()`.

### 2.2.11 Clean up your objects

Even though VB has a built in garbage collector to reclaim resources from un-referenced objects and variables, it is good practice to destroy your objects as soon as possible. This improves scalability by making it possible for the system to re-use objects.

Therefore, close file handles, database connections and recordsets explicitly before destroying the objects using the following syntax.

```
set oObjectName = nothing
```

### 2.2.12 Transactions (ADO)

VB does not really have native support for transactions. Transactions are usually controlled directly via ADO or COM+. COM+ transactions are out of the scope of this document, however an example of ADO transactions is displayed below.

```
...
oConnShadow.BeginTrans
if not ProcessSingleImportFile(oFile) then
  LogMessage "Import order failed, rolling back transaction."
  oConnShadow.RollbackTrans
else
  LogMessage "Importing orders succeeded, committing transaction"
  '** Check if it actually commits the transactions
  on error resume next
  oConnShadow.CommitTrans
  if err then
```



```

        LogMessage "An error occurred while committing the " & _
            transaction: " + err.Description
    end if
    on error goto 0
end if
...

```

Please keep in mind that ADO transactions do not always replace T-SQL transactions. T-SQL transactions control everything that happens in one stored procedure, while ADO transactions can be used to group a number of SP calls into one big transaction.

Transactions are important for data integrity purposes, **USE THEM!**

### 2.2.13 Reading & writing configuration settings

Unless there is a good reason not to, settings should be read from the registry. Don't use the built in VB registry functions as these don't work well when invoked from an ASP page. The Windows Scripting Host ships with a com object that can manipulate the registry. An example is given below.

```

...
sKey = "HKEY_LOCAL_MACHINE\Software\Demachina\Perfy\DSN"
set oShell = CreateObject("WScript.Shell")
sDSN = oShell.RegRead(sKey)
set oShell = nothing
...

```

## 2.3 C#

This chapter outlines the conventions for C#. Additional recommendations, for when C# is used as the programming language in ASP.NET, are made in 2.6.

### 2.3.1 Copyright message & page header

The page header is almost identical to the one described in 2.1.1.

```

/*****
<Solution Name>

Copyright 2005, Demachina
www.demachina.com

All rights reserved

Document author: Jeroen Ritmeijer

FILENAME      : DataManagerObject.cs
PURPOSE       : Implementation for AbstractDataManagerObject
MODULE        : Generic data objects
DEPENDENCIES  :

TO-DO         : - Error handling
               - Parameter validation

KNOWN ISSUES  : -
TAB SIZE      : 4
HISTORY       : 16/01/2004 - JR - Created file
*****/

```

### 2.3.2 Document the code

As described in 2.1.2 with the following addition:

Visual Studio .Net has the capability to create Task or 'To do' lists based on comments in the code. Add '// TODO: *Description*' and '// HACK: *Description*' to automatically populate these lists.

An example of the typical level of documentation required is listed below.

```
/// <summary>
/// Return the translation for the specified key and locale.
/// </summary>
/// <param name="localeID">The locale for which to get the translation</param>
/// <param name="primaryID">Primary ID for identifying a translation
/// such as 'WelcomeScreen'. This parameter is case independent.</param>
/// <param name="secondaryID">Secondary ID for identifying a translation
/// such as 'LoginError'. This parameter is case independent.</param>
/// <returns>The translation of null if the translation cannot be
/// found.</returns>
public static String Translate(String localeID, String primaryID,
    String secondaryID)
{
    try
    {
        // ** Validate parameters
        if(localeID == null)
            throw (new ArgumentNullException("localeID"));
        if(primaryID == null)
            throw (new ArgumentNullException("primaryID"));
        if(secondaryID == null)
            throw (new ArgumentNullException("secondaryID"));

        // ** Fix up parameters
        if(primaryID.Length > 20)
            primaryID = primaryID.Substring(0,20);
        if(secondaryID.Length > 20)
            secondaryID = secondaryID.Substring(0,20);

        // TODO: Optimise performance by caching hashtable for last accessed
        // LocaleID make sure this cached hashtable is removed when the
        // translations are cleared / deleted / reloaded.
        Hashtable tempHash = null;

        // ** Get the hashtable for the specified locale
        tempHash = (Hashtable) translationHash[localeID];

        // ** Return the description for the specified key
        if(tempHash != null)
        {
            return (String) tempHash[primaryID.ToLower() + keyDelimiter +
                secondaryID.ToLower()];
        }
        else
        {
            return null;
        }
    }
    catch(Exception ex)
    {
        // ** Handle the exception in the centralised exception handler
        ExceptionManager.Publish(ex);
        // ** Enrich and rethrow it to the calling method.
        String errorString = String.Format("An error occurred while retrieving a
            translation.\nlocaleID: {0}\nprimaryID: {1}\nsecondaryID: {2}",
                localeID, primaryID, secondaryID);
        throw (new TranslationException(errorString, ex));
    }
}
```

### 2.3.3 Documenting functions

As described in 2.1.3 with the following changes. .NET languages also define how functions are to be commented in order to automatically generate the documentation.

This entire process is well defined in .Net literature and out of the scope of this document. For more information see <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcoriXMLDocumentation.asp?frame=true>

```
/// <summary>
/// Read the content of a parameter back. This is usually done to read the
/// primary key back after executing a stored procedure
/// </summary>
/// <param name="parameterName">The name of the parameter to read.
/// <see cref="AddParameter"/> for more information.
/// </param>
/// <returns>A System.Object containing the parameter value.</returns>...
```

### 2.3.4 Formatting Code

As described in 2.1.5 with the following additions:

Visual Studio .NET supports so called *Code Regions*, which can be used to collapse large sections of code in order to keep things readable.

Group properties, methods and constructors into groups using the '#region description' and '#endregion description' statements as shown in the screenshot below.

```
public class DataManagerObject : MCRL.Data.AbstractDataManagerObject
{
    #region Private properties
    ...
    #endregion

    #region Constructors
    ...
    #endregion

    #region Public methods
    ...
    #endregion

    #region Private methods
    ...
    #endregion

    #region Public properties
    ...
    #endregion

    #region IDisposable Members
    ...
    #endregion
}
```

Typical look of a class when all regions are collapsed.

### 2.3.5 Naming conventions

The naming convention for C# has been clearly defined by Microsoft. A full discussion is out of the scope of this document, but a summary is provided below.

Identifier	Case	Example
Class	Pascal	AppDomain

Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChanged
Exception class	Pascal	WebException Note Always ends with the suffix Exception.
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable Note Always begins with the prefix I.
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance field	Camel	redValue Note Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	RedValue Note Rarely used. A property is preferable to using a public instance field.

In short, always use *Pascal case* except for Class or function variable names, which use *Camel case*.

For more information see

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnamingguidelines.asp>

### 2.3.6 Object Orientation

All .NET languages are fully object oriented, make use of it.

### 2.3.7 Error handling

C# supports structured exception handling via the 'try...catch' mechanism. Please use this mechanism extensively.

Additionally, use the Exception *Application Block* as published by Microsoft at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp>. Familiarise yourself with this technology and use it.

An example of what a typical error handler looks like is given below.

```
try
{
```

```
// ... Actual implementation
}
catch(Exception ex)
{
    // ** Handle the exception in the centralised exception handler
    ExceptionManager.Publish(ex);
    // ** Enrich and rethrow it to the calling method.
    String errorString = String.Format("An error occurred in FillDetails.\n" +
        " - customerID: {0}\n" +
        " - sessionID: {1}\n" +
        " - itemsFound: {2}\n",
        customerID, sessionID, itemsFound);
    throw (new WebClientApplicationException(errorString, ex));
}
```

A generic guideline for Exception Management in .NET is available from <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp>

### 2.3.8 Concatenating strings

Keep in mind that concatenating many strings in a loop can become very slow due to constant memory re-allocation requests. Either use a chunking technique or a smart string formatter to create large strings.

Luckily the .NET framework provides helper classes and functions for both techniques namely the *StringBuilder* class and the *Format* method.

```
...
// Open recordset as Data reader
...
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("<TABLE>");
// ** Iterate through a large list and perform a lot of concatenations
while(rdr.Read())
{
    stringBuilder.Append("<TR><TD>" + rdr.GetValue(0).ToString() +
        "</TD></TR>");
}
System.Windows.Forms.MessageBox.Show(stringBuilder.ToString());
```

*Example of using the StringBuilder class*

```
...
// Open recordset as Data reader
...
// ** Iterate through a large list and perform a lot of concatenations
while(rdr.Read())
{
    DebugTxt(String.Format("<TR><TD>{0}</TD><TD>{1}</TD></TR>",
        rdr.GetValue(0).ToString(), rdr.GetValue(1).ToString()));
}
```

*Example of using the Format method.*

### 2.3.9 Call *Dispose()* when required

.NET languages use a garbage collector to clean up after itself, however certain objects, especially those that claim resources or use unmanaged code need to be freed as soon as possible.

Typically these classes implement the *IDisposable* interface. When using an object that implements this interface, call the *Dispose()* method to clean up the resources. DO NOT FORGET!!!!!!

### 2.3.10 Caching

The .NET framework offers a built in caching mechanism in the *System.Web.Caching.Cache* namespace. Use this mechanism to store 'expensive to access' data such as database records that are read frequently, but changed infrequently.

Alternatively, for very simple scenarios, or for those scenarios where you need more control over the cached data, use *Static* member variables to store data and share it between instances of a class.

### 2.3.11 Transactions (ADO.NET)

C#, as a language, does not really have native support for transactions. Transactions are usually controlled directly via ADO.NET or the *System.EnterpriseServices.ServicedComponent* class. *ServicedComponents* are out of the scope of this document, but an example for ADO.NET transactions is provided below.

```
...
OracleTransaction myTrans = conn.BeginTransaction();

try
{
    OracleCommand cmd = new OracleCommand();
    cmd.Connection = conn;
    cmd.Transaction = myTrans;
    cmd.CommandText = "PRODUCTS_PKG.UpdateProductByID";
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add(new OracleParameter("Product_ID_IN",
        OracleType.Number)).Value = txtUpdateRecordID.Text;
    cmd.Parameters.Add(new OracleParameter("ProductName_IN",
        OracleType.VarChar)).Value = txtUpdateRecordName.Text;
    cmd.ExecuteNonQuery();

    cmd = new OracleCommand();
    cmd.Connection = conn;
    cmd.Transaction = myTrans;
    cmd.CommandText = "PRODUCTS_PKG.InsertProduct";
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add(new OracleParameter("ProductName_IN",
        OracleType.VarChar)).Value = txtInsert.Text;
    cmd.Parameters.Add(new OracleParameter("Product_ID_OUT",
        OracleType.Number)).Direction = ParameterDirection.Output;
    cmd.ExecuteNonQuery();

    // cause error
    throw(new Exception("Random exception to test transactions"));

    myTrans.Commit();
}
catch(Exception ex)
```

```
{
    MessageBox.Show(ex.ToString());
    myTrans.Rollback();
}
finally
{
    conn.Close();
}
...
```

Please keep in mind that ADO transactions do not always replace T-SQL or PL/SQL transactions. T-SQL transactions control everything that happens in one stored procedure, while ADO transactions can be used to group a number of SP calls into one big transaction.

Transactions are important for data integrity purposes, **USE THEM!**

## 2.4 JavaScript

This chapter outlines the conventions for Jscript, JavaScript and ECMAScript. Additional recommendations, for when JavaScript is used to validate forms are made in *2.9.6 Validate forms using JavaScript*.

Please note that we do not use JavaScript for server side ASP programming or for writing stand alone scripts (.wsf) files. The convention is to use VBScript for those tasks.

### 2.4.1 Copyright message & page header

Only include a page header and copyright message in stand alone .js files. Don't include it when the JavaScript is embedded inside an ASP or HTML file as those files already have their own page header and copyright message.

```
/******
'
'                               <Solution Name>
'
'                               Copyright 2005, Demachina
'                               www.demachina.com
'
'                               All rights reserved
'
'                               Document author: Jeroen Ritmeijer
'
' FILENAME      : orders.js
' PURPOSE       : Check if any order...
' MODULE        : Order imports, For more info see: specs.doc
' DEPENDENCIES  : javascript/products.js
'
' TO-DO         : -
' KNOWN ISSUES  : -
' TAB SIZE      : 4
' HISTORY       : 04/03/2002 - JR - Created file
'               : 07/03/2002 - GT - Performed code review
' *****/
```

### 2.4.2 Document the code

As described in 2.1.2.

### 2.4.3 Documenting functions

As described in 2.1.3.

### 2.4.4 Formatting Code

As described in 2.1.5 with the following additions:

**Location:** Unless there is a good reason not to do so, place JavaScript inside the HTML <HEAD> element as this is the recommended location.

**Breaking up lengthy statements:** Breaking up long statements into multiple lines makes your code easier to read, both online and when printed. The following example breaks a statement into three lines:

```
...
if (isError)
{
    LogMessage ("An error occurred while updating the order " +
               "status for OrderID " + orderID +
               ". The error is: " + error);
    success = false;
...

```

Some limitations exist as to where a line break can be used, such as in the middle of an argument name. You can break up an argument list, but the individual names of the arguments must remain intact. Additionally you cannot have multi line string definitions. Close the string, add a '+' followed by the line break.

**End of line:** End all statements with ';' although JavaScript is very forgiving, certain utilities such as JavaScript code compressors are not.

**Indentation:** Wars have been fought over this one. There are two movements that have been battling with each other about where to place curly braces '{' in C like languages such as Java and JavaScript.

*Movement 1:*

```
function someName()
{
    if(..)
    {
    }
}
```

*Movement 2:*

```
function someName() {
    if(..) {
    }
}
```

Our standard is 'Movement 1', with the exception that when you are working on an existing module that follows a different standard, you should follow that standard to ensure things don't become a mess.



#### 2.4.5 Naming conventions

The naming convention JavaScript resembles the convention for Java (see 2.10.6). A summary is provided below.

Identifier	Case	Example
Class	Pascal	CStringBuilder
Method	Camel	toString
Property	Camel	backColor
Variable	Camel	redValue

JavaScript supports the encapsulation of functions into classes. Our standard is to prefix all member function definitions with the class name followed by an underscore as the following example will make clear:

```
// Class definition, full comments omitted for clarity
function CProducts
{
    // ** Methods
    this.render = CProducts_Render;
    this.load   = CProducts_Load;
}

function CProducts_Render()
{
    ...
}

function CProducts_Load()
{
    ...
}
```

Please note that the actual member function (when it is part of a class) does not contain the class name.

#### 2.4.6 Declaring variables

As described in 2.1.9. with the following addition:

There is no equivalent to VB's 'option explicit' statement in JavaScript.

#### 2.4.7 Error handling

There are a number of ways to handle errors in JavaScript depending on the platform you are developing for.

If you are lucky you are developing for IE4+ and NS6+ because then you can use structured exception handling using "try {} catch {} [finally {}]" as displayed in the example below.

```
try
{
    WScript.echo ("Starting");
    var i = a.x;
    WScript.echo ("This line is never displayed");
}
```

```
catch(e)
{
    WScript.echo ("Error number: " + e.number);
    WScript.echo ("Error description: " + e.description);
}
```

In all other cases you have to either code without errors or use the 'window.onerror' event handler as illustrated in the following example.

```
function errorHandler(msg,url,line)
{
    displayFriendlyError(msg + "\n" + url + "\n" + line);
    return true;
}

function causeError()
{
    var i = a.x;
    // ** The following line is never executed
    alert('post error');
}

window.onerror = errorHandler;

causeError();
```

The *onerror* event fires when a scripting error occurs, for example, when there are security problems during the loading process or just bad coding.

The *line* parameter is an integer specifying the line number at which the error occurred, *url* is a string specifying the URL of the document containing the error, and the optional *message* is a message describing the error.

This event will not bubble. Events that do not bubble can only be handled on the individual object that fired the event.

One of the very few useful things that can be done with this event handler is to redirect the user to an error page that may or may not take corrective action or just notify the administrators and display a friendly message.

#### 2.4.8 Validating forms

Validating forms using JavaScript is described in detail in 2.9.6.

#### 2.4.9 Concatenating strings

Concatenating a large number of strings is slow. Use the CString class (Available on request) to speed up JavaScript string processing by an order of magnitude.

An example is given below.

```
var html = new CString("");
for (var i=0; i<products.Count(); i++)
{
    html.Add("A lot of HTML");
    html.Add("A lot of HTML");
    html.Add("A lot of HTML");
    html.Add("A lot of HTML");
    html.Add("A lot of HTML");
    html.Add("A lot of HTML");
}
return html.toString();
```

#### 2.4.10 Including files

Including JavaScript files is easy as this is supported by HTML. It will be no surprise that the syntax is:

```
<SCRIPT LANGUAGE="JavaScript" SRC="javascript/library.js"></SCRIPT>
```

#### 2.4.11 Clean up your objects

Even though JavaScript has a built in garbage collector to reclaim resources from un-referenced objects and variables, it is good practice to destroy your objects as soon as possible. This will result in a lower memory footprint in the customer's browser.

The JavaScript syntax to dereference an object is

```
objectName = null;
```

#### 2.4.12 Reading & writing configuration settings

JavaScript in the browser can not directly access either the registry or a configuration database, therefore configuration settings need to be passed by JavaScript using ASP. Do not hardcode anything.

## 2.5 ASP

Our convention is to program ASP exclusively in VBScript in combination with COM objects. Please see section 2.2 for more information on VBScript coding conventions.

Please note that we use different standards for ASP.NET.

#### 2.5.1 Copyright message & page header

Because an ASP page can take parameters from a number of sources (Cookies, Form, querystring etc) we have added a number of fields to the page header. An example is given below.

```
*****
'
'                               <Solution Name>
'
'                               Copyright 2005, Demachina
'                               www.demachina.com
'
'                               All rights reserved
'
'                               Document author: Jeroen Ritmeijer
'
' FILENAME      : order.asp
' PURPOSE       : Check if any order...
' MODULE        : Order imports, For more info see: specs.doc
' DEPENDENCIES  : inc_database.asp, inc_security, styles.css
'
' Input parameters
```

```
' COOKIES      : -
' QUERYSTRING : action - contains the action to perform, either
'                  update', 'add', 'new', 'edit' or 'delete'.
' FORM        : id      - The primary key of the user to operate on
'
' Page returns
' COOKIES      : -
' DATA        : The 'edit order' page
'
' TO-DO        : Make transactional
' KNOWN ISSUES : -
' TAB SIZE     : 4
' HISTORY      : 25/02/2002 - JR - Created file
' *****
```

### 2.5.2 Document the code

As described in 2.1.2.

### 2.5.3 Documenting functions

As described in 2.1.3.

### 2.5.4 Formatting Code

As described in 2.4.4 with the following additions:

**Interleaving ASP & HTML:** Don't switch between HTML and ASP too often as this results in unreadable code. It also works against the '*inlining code*' rule (see below)

*How **not** to do it:*

```
...
<% oProducts.Load "[...]" %>
<BODY>
    product list<BR>
    <% for each oProduct in oProducts %>
        Productname: <% =oProduct.Name %><BR>
    <% next %>
</BODY>
<% set oProducts = nothing %>
...
```

*The proper way to do it:*

```
...
<% function PrintProductList()
    dim oProducts
    set oProducts = OpenProducts()
    for each oProduct in oProducts
        response.write("Productname: " & oProduct.Name & "<BR>")
    next
    set oProducts = nothing
end function
%>
<BODY>
    <% PrintProductList() %>
</BODY>
...
```

**Positioning functions:** Don't scatter your function definitions all over the ASP page. Place them at the beginning of the file in a group, or even better if there

are more than a small number of functions available, place them in an include file or COM object. This makes it much easier to share code between channels.

**Inlining code:** Don't inline code too much in your asp page. If you have more than a small number of lines of code consider putting it in a function.

### 2.5.5 Error handling

Considering our standard is to write ASP code in VBScript all rules defined in 2.2.7 apply. The difference however is deciding what you do with the error.

### 2.5.6 Presenting errors to the user

Because ASP represents the user interface we have to make sure the user is presented with a 'user friendly' page that explains something has gone wrong as described in 2.1.8.

Libraries for this purpose are company specific and will need to be developed by a core team. Create these based on the guidelines below.

The proper thing to do is to create one generic error.asp page, which has the following functionalities:

It should accept the following parameters:

1. **ErrorNumber:** corresponds to err.number. The documentation should contain a full list of error messages.
2. **Severity:** Based on the severity number we can undertake a different action or display a different message. We suggest the following severity levels:
  1. **System error (level 1-10):** A system error, something has gone wrong that caused the core of the system to fail. Errors like 'database not available' fit into this category. Whenever an error like this occurs an e-mail is submitted by the server to the administrator and a message indicating that a sever error has occurred and that the administrative staff has been notified should be displayed to the user.
  2. **Unexpected error (level 11-20):** An error has occurred in a place where an error should not occur ( $1+1 \neq 2$ ). Whenever an error like this occurs an e-mail is submitted by the server to the administrator and a message indicating that a sever error has occurred and that the administrative staff has been notified should be displayed to the user. The user is asked to 'try again'.
  3. **User error (level 21-30):** An error caused by the user, either by not using a proper browser, filling in bogus information, tempering with cookies or URL parameters. A message should be displayed indicating that a user error has occurred. This kind of error should not be raised when an *expected* error occurs like 'invalid username or password' when logging in. These kinds of errors should be handled in the page directly.
3. **UserMsg:** an optional friendly error message describing the user what

has happened. If this parameter is omitted one of the messages described in *severity* will be displayed.

4. **Error:** Full error message if available that will be submitted by mail to the administrative staff.

Please remember that the maximum URL length in IE is roughly 2000 characters. Do not exceed this length because explorer will simply not navigate to the specified page. Do not forget to URL-encode the parameters because an error message may contain codes that are not allowed in a URL.

When an error occurs with a severity level between 1 and 20 an error message needs to be submitted to an administrative e-mail account. This messages should include:

- A timestamp
- All parameters described in the previous paragraph
- A full dump of request.servervariables

Whenever an error occurs, either in client side JavaScript or in server side asp code, the customer should be redirected to this page. **Make sure this page is displayed in a visible frame.** If an error occurs in one of the data frames it doesn't make sense to display the error there. There should always be one designated frame / window for displaying errors.

#### 2.5.7 Separate presentation from business logic.

As described in 2.1.14 and 2.5.4

#### 2.5.8 Validate input parameters

Always check if URL or FORM parameters that are expected by an asp file to be available actually are available and are of the correct type. If not, the user has been tempering with the URL and should be redirected to the error page with a *User Level* severity (See 2.5.6)

If you create a dynamic SQL statement to retrieve a product list, make sure you filter the string you pass into the free text indexer for reserved characters and keywords. The list of reserved keywords for the SQL server Full text indexer is:

- AND
- OR
- Whatever your string delimiter is to build the SQL statement.

Also make sure the string does not only consist of words that are on the ignore list. No matter what, catch any errors that occur and display a friendly error message for the customer. Optionally log these invalid keywords to the database so the website operator can get insight into products people are searching for that cannot be found.

#### 2.5.9 Including files

Re-use code by including external files using the following syntax:

```
<!-- #include file="includes/renderdata.asp" -->
```

Never give include files the extension .inc as those are uninterpreted files that can be downloaded by potential hackers.

#### 2.5.10 Use a proper default document

The convention is to use default.asp for the first web page that is displayed to the user. This is the same file that IIS uses by default. Do not use names such as home.asp, welcome.asp or main.asp.

#### 2.5.11 Transactions

COM+ transactions are completely out of the scope of this document, however some recommendations on how to use the ASP native transaction capabilities are made below.

When a page performs multiple database operations (excluding reads) it is required that you wrap everything in one transaction. You can either do this by using ADO transactions (see 2.2.12) or even better use ASP's built in capabilities. At the top of the page add the following line:

```
<% @transaction = REQUIRED %>
```

If an error occurs anywhere on the page all database changes are automatically rolled back.

More information can be found in the various IIS tutorials, the MSDN library and ASP books.

#### 2.5.12 Are HTTP caching headers being used?

By properly manipulating the headers of a web page, proxy servers and browsers can be instructed to not go back to the web server for each request, but rather return the contents from a local cache. This improves both the response time for the customer and reduces the stress on the web server. This is one of the best optimisations you can make to your site and you basically get it for free because it is part of the HTTP protocol.

Only set these headers for pages that are requested most frequently. The settings should be specified in the global.asa to make it easy to switch between caching (release version of the website) and no caching (development version of the site).

Each cache configuration consists of two parameters: one specifies the 'CacheControl', which act as an *enable* or *disable* flag ('public' means cache if you can, 'private' means never cache), the other specifies the cache timeout. This timeout is stored in the 'expires' property of the ASP response object and specifies in minutes how long the file should be cached before requesting it again from the server.

e.g.

```
/** Category cache settings
Application("CacheCategoriesHeaderControl") = "public"
Application("CacheCategoriesHeaderExpires") = 60
/** Products cache settings
Application("CacheProductsHeaderControl") = "public"
```

```
Application("CacheProductsHeaderExpires") = 60
' ** Basket cache settings (do not cache)
Application("CacheBasketHeaderControl") = "private"
Application("CacheBasketHeaderExpires") = -1000
```

This makes it possible to put the following code in pages like /shop/getproductdata.asp:

```
Response.CacheControl = Application("CacheProductsHeaderControl")
Response.Expires = Application("CacheProductsHeaderExpires")
```

Please note that these settings are just hints to the several caching mechanisms, it does not guarantee that the file ends up being cached.

### 2.5.13 Are global variables defined in the global.asa?

Always declare variables like e-mail addresses, caching information (see 2.5.12 *Are HTTP caching headers being used?*) and other variables, that might be different when running the site in *release* and in *debug* mode, as application variables.

This makes it much easier to quickly make changes without figuring out which files need to be changed manually.

### 2.5.14 Reading & writing configuration settings

As described in 2.2.13 with the following addition:

Read all parameters in the application\_onstart event in the global.asa and store them in application variables.

## 2.6 ASP.NET

### 2.6.1 Copyright message & page header

Because an ASP.NET page can take parameters from a number of sources (Cookies, Form, querystring etc) we have added a number of fields to the page header.

This page header should be copied to the .aspx AS WELL AS the 'code behind' file.

An example is given below.

```
<%
/*****
                                     <Solution Name>

                                     Copyright 2005, Demachina
                                     www.demachina.com

                                     All rights reserved

                                     Document author: Jeroen Ritmeijer

FILENAME       : order.aspx
PURPOSE        : Check if any order...
MODULE         : Order imports, For more info see: specs.doc
*****/
```



```

DEPENDENCIES : inc_database.asp, inc_security, styles.css

Input parameters
COOKIES      : -
QUERYSTRING  : action - contains the action to perform, either
                  update', 'add', 'new', 'edit' or 'delete'.
FORM         : id      - The primary key of the user to operate on

Page returns
COOKIES      : -
DATA         : The 'edit order' page

TO-DO        : Make transactional
KNOWN ISSUES  : -
TAB SIZE     : 4
HISTORY      : 30/01/2004 - JR - Created file
***** /
%>

```

Note that the entire header is enclosed by ASP.NET tags (<% %>) to make sure it is not send repeatedly to the browser.

## 2.6.2 Separate presentation from business logic

As specified in 2.1.14 with the following addition.

ASP.NET uses so called 'code behind' files to separate the logic from the presentation. Make use of this capability and avoid putting any logic directly in the ASP.NET (.aspx) file.

The 'code behind' file can even pass output to the ASP.NET page by making use of <asp:Label> webcontrols.

A sample is given below to illustrate this.

```

<%@ Page CodeBehind="Welcome.aspx.cs" Language="c#"
Inherits="GenericPSA.Welcome" %>
...
<BODY>
    <asp:Label ID="LoginError" RUNAT="server" VISIBLE="false">
    </asp:Label>
...

```

*Fragment of an .aspx page*

```

...
protected System.Web.UI.WebControls.Label LoginError;

private void Page_Load(object sender, System.EventArgs e)
{
    if(GetStatus() == false)
    {
        LoginError.Text = "Your account is disabled";
        LoginError.Visible = true;
    }
...

```

*Fragment of the accompanying 'code behind' file.*

## 2.7 Microsoft T-SQL

T-SQL is the language used by Microsoft SQL server for programming stored procedures. While programming in T-SQL Microsoft's Books Online (Installed with SQL Server) can be an extremely helpful resource.

### 2.7.1 Copyright message & page header

Listed below is an example of an actual T-SQL page header.

```

/*****
<Solution Name>

Copyright 2005, Demachina
www.demachina.com

All rights reserved

Document author: Jeroen Ritmeijer

FUNCTION      : LoadOrders
PURPOSE       : Import orders from the live database on the Transactional
                website to the store that this database is hosted in. For more
                information see 'DB Documentation.doc' and
                'Administrative module for retrieving order information'
MODULE        : Order import / export
DEPENDENCIES  : Admin table, xportOrderHeaders table
PARAMETERS    : @FromShadowDatabase bit : Load from shadow database (1) or
                from live server (0 or not specified)

RETURNS       : -
TO-DO         : -
KNOWN ISSUES  : - This version can only handle one 'change' at a time. So an
                order cannot have a new delivery address and a new delivery
                slot at the same time. This scenario is unlikely to happen in
                the real world anyway.
                - Depending on what happens in Symbol's triggers, their errors
                might not be detected
                - T/SQL Error handling is horrendous, there are two instances
                in this SP where we are checking for errors, but they will
                not always be reported by MS-SQL even if an error occurs.
                These instances are clearly marked in the code.

TAB SIZE      : 4
HISTORY       : 20/11/2001 - JR: Rewrite to make the lot transactional
                01/03/2002 - JR: Added optional parameter
                to allow OrderLoads from the shadow DB.
                Changed error handling to point to a separate
                error handler DB
                07/03/2002 - JR: Added code to delete the source orders when we
                are importing from the shadow database.
                15/03/2002 - MC: Added DeliveryPrice to Delivery Slot update
                *****/

```

### 2.7.2 Documenting stored procedures & functions

As described in 2.1.2. and 2.1.3 with the following addition:

A stored procedure is a function and therefore does not require a separate function header as all information is already made available in the page header.

### 2.7.3 Formatting Code

As described in 2.4.4 with the following additions:

Use the SQL Query Analyzer for coding, not the simple window in the enterprise manager as that one uses a proportional font that messes up existing indentation.

Break up long statements and distribute them over multiple lines as displayed in the example below.

```
...
    SELECT Modules.*, UserModules.UserID
        FROM Modules WITH(NOLOCK)
            LEFT JOIN UserModules WITH(NOLOCK, FASTFIRSTROW) ON
                Modules.ModuleID = UserModules.ModuleID AND
                UserModules.UserID = @UserID
        ORDER BY Modules.Name
...
```

Although the syntax is different, indent 'BEGIN...END' blocks in the same way as in JavaScript (See 2.4.4). Listed below is a small example:

```
...
    IF SomeCondition
    BEGIN
        SELECT * FROM Stores
    END
    ELSE
    BEGIN
        SELECT * FROM Users
    END
...
```

As can be seen in the two examples above, the convention is to capitalise all T-SQL reserved keywords.

Begin each separate word in a field name with a capital letter, as in FindLastRecord and RedrawMyForm.

For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" will lead to confusion.

#### 2.7.4 Naming conventions

Do not use spaces within the name of database objects, as spaces confuse front-end data access tools and applications.

Make sure to not use any reserved words for naming database objects, as that can lead to some unpredictable situations. To get a list of reserved words for Microsoft SQL Server, search Books Online for 'Reserved keywords'

##### 2.7.4.1 Tables

Tables represent the instances of an entity. For example, you store all your customer information in a table. Here, 'customer' is an entity and all the rows in the customers table represent the instances of the entity 'customer'. So, name your table using the entity it represents, 'Customer' (singular, not plural).

Examples of valid table names are:

- 'Customer' for your customer table.
- 'CustomerOrder' for your order table. (don't use 'order' as that is a reserved keyword)
- 'ErrorMessage' for your error messages table.

This is a more natural way of naming tables, when compared to approaches which name tables as tblCustomers or tbl\_Orders. Further, when you look at your queries it's very obvious that a particular name refers to a table, as table names are always preceded by the FROM clause of the SELECT statement.

If your database deals with different logical functions and you want to group your tables according to the logical group they belong to, prefix your table name with a two or three character prefix that can identify the group.

For example, your database has tables which store information about Sales and Human resource departments, you could name all your tables related to Sales department as shown below:

- SL\_NewLead
- SL\_Territory
- SL\_TerritoryManager

You could name all your tables related to Human resources department as shown below:

- HR\_Candidate
- HR\_PremierInstitute
- HR\_InterviewSchedule

This kind of naming convention makes sure all the related tables are grouped together when you list all your tables in alphabetical order. However, if your database deals with only one logical group of tables, don't use this naming convention.

#### 2.7.4.2 Columns

Columns are attributes of an entity, that is, columns describe the properties of an entity. So, let the column names be meaningful and natural.

Examples of valid Names are:

- ID
- FirstName
- Address

As you can see, we do not use underscores in column names and capitalize the first character of each word. Do not prefix the name of the column with the table name (CustomerID, CustomerFirstName, etc) as we already know what table we are in.

#### 2.7.4.3 Stored procedures

Stored procedures always do some work for you, they are action oriented. So, let their name describe the work they perform and use a verb to describe the work.

Based on the type of function they perform, prefix the name with the action type (Select, Insert, Update or Delete). Note that a stored procedure does not always fit in one of these action categories, in that case do not prefix the stored procedure and just give it a logical name.

Examples of valid stored procedure names:

- DeleteUser
- SelectUsers
- ValidateUserForModule

Examples of invalid stored procedure names:

- GetUsers (use SelectUsers in stead)
- Validate\_User\_For\_Module (Don't use underscores)

As explained above in the case of tables, you could use a prefix to group stored procedures depending upon the logical group they belong to. For example, all stored procedures that deal with 'Order processing' could be prefixed with ORD\_ as shown below:

- ORD\_InsertOrder
- ORD\_InsertOrderDetails
- ORD\_ValidateOrder

If you are using Microsoft SQL Server, never prefix your stored procedures with 'sp\_', unless you are storing the procedure in the master database. If you call a stored procedure prefixed with sp\_, SQL Server always looks for this procedure in the master database. Only after checking in the master database (if not found) it searches the current database.

#### 2.7.4.4 Indexes

Although you never reference named indexes directly in your T-SQL code, it is good practice to give them a useful name. When using SQL Enterprise manager to add an Index, a default index name is provided using the following convention:

IX\_<table name>

Our convention is to add the field name(s) to this so it will look like

IX\_<table name>\_<field name>

A possible Index name for the Customer.ID field is:

IX\_Customer\_ID

### 2.7.5 Use Named arguments in complex SP calls

A simple trick to make your code more readable is to use named arguments when calling stored procedures with a less than obvious parameter list.

```
...
EXEC SharedComponents..LogErrorToTable
    @ErrorNumber = 1004,
    @SeverityLevel = 2,
    @FriendlyDescription = @ErrorString,
    @FullDescription = @sql,
    @Source = 'LoadOrders/@Status=2032/UpdateOrderStatus=2',
    @ErrorGroup = @DefaultErrorGroup
...
```

Only use it when it makes sense, don't use it when calling:

```
EXEC SelectProductsByStoreID 1
```

It is pretty obvious that '1' represents the StoreID in this call.

### 2.7.6 Minimizing Code

Stored procedures should be used for data storage and retrieval only. This has a number of advantages:

1. **Scalability:** Databases are very difficult to scale beyond one machine and are therefore the main bottleneck in modern e-commerce applications. By minimising the complexity in a stored procedure the CPU is offloaded. The actual business logic should go in COM components that are easy to distribute over multiple machines.
2. **Portability:** Less code means less chance of using database specific and proprietary code. This makes it easier to port the database to a different platform such as Oracle, DB2 or Informix.
3. **Simplicity:** Stored procedures are difficult to debug. Less code means less chance of bugs.

### 2.7.7 Error handling

Error handling is one of the weakest points of T-SQL. Because it is so error prone and user unfriendly it is often 'forgotten' by T-SQL programmers, however in order to build a robust solution it is absolutely essential to make use of it. As always most of the details can be found in MS-SQL Books Online (BOL), however a short summary is supplied below.

As BOL will tell you, the connection wide @@ERROR variable always contains the status of the previously executed statement. That statement can be a complex INSERT or a simple PRINT command. Therefore it is essential that you 'save' the value of @@ERROR to a local variable immediately after executing a statement that potentially might fail.

An example of proper error handling and logging is provided below.

```
...
UPDATE StoreOrderHeader
SET MStoreOrderHeader.DeliverySlotStart =
    #OrderHeader.DeliverySlotStart,
    StoreOrderHeader.DeliverySlotEnd = #OrderHeader.DeliverySlotEnd,
    StoreOrderHeader.DeliveryCharge = #OrderHeader.DeliveryCharge
```

```

FROM StoreOrderHeader INNER JOIN #OrderHeader
    ON StoreOrderHeader.OrderID = #OrderHeader.OrderID
WHERE #OrderHeader.OrderID = @OrderID
SET @err = @@ERROR
IF (@err <> 0)
BEGIN
    SET @CommitTransaction = 0
    SET @ErrorString = 'Error setting the status flag to ''synced'' on ' +
        'the main database for OrderID: ' + CAST(@OrderID as VARCHAR(255))
    EXEC SharedComponents..LogErrorToTable
        @ErrorNumber = 1004,
        @SeverityLevel = 2,
        @FriendlyDescription = @ErrorString,
        @FullDescription = @sql,
        @Source = 'LoadOrders/@Status=2032/UpdateOrderStatus=2',
        @ErrorGroup = @DefaultErrorGroup
END
...

```

Error handling in T-SQL goes hand in hand with transactions, for more information see 2.7.8.

### 2.7.8 Transactions

Transactions can be a lot more complicated than people think. Not everything is done for you automatically and you have to be really careful about what you are doing. Again, BOL to the rescue, but for clarity an example is given below.

```

CREATE PROCEDURE DeleteUser
    @UserID      int
AS
    DECLARE @err int

    BEGIN TRANSACTION

    --** Delete module access for this user.
    DELETE FROM UserModule
        WHERE UserID = @UserID
    SET @err = @@error

    --** Did an error occur?
    IF @err = 0
    BEGIN
        --** Delete user.
        DELETE FROM User
            WHERE UserID = @UserID
        SET @err = @@error
    END

    --** Did an error occur somewhere in this procedure?
    IF @err = 0
    BEGIN
        COMMIT TRANSACTION
    END
    ELSE
    BEGIN
        ROLLBACK TRANSACTION
        --** Insert proper error handling here.
        print 'error: ' + CAST(@err AS VARCHAR(255))
    END
END

```

In line with 2.7.7 the @@ERROR variable is immediately saved after a statement is executed. Default T-SQL behaviour is to manually check for errors and based on the error codes to roll back or commit a transaction. This

behaviour can be changed (please don't) by calling the 'SET XACT\_ABORT ON' function. As always, for detailed information see BOL.

### 2.7.9 Take care of locking

Locks are a natural thing in database environments as every experienced database programmer will tell you. The thing to be careful of is to not end up in a deadlock situation. Deadlocks can easily be detected by putting realistic stress on the solution / database and checking for #1205 errors. The best thing to do is to prevent deadlocks altogether. Most deadlocks can be avoided by performing dirty reads as shown in the following example.

```
CREATE PROCEDURE CustomerLogin
    @UserID as varchar(128),
    @Password as varchar(32)
BEGIN
    SELECT CustomerID
    FROM Customer WITH (NOLOCK)
    WHERE UserID = @UserID AND
           Password = @Password
```

The example above performs a read, but ignores any kind of locks on the table that might be caused by a Customer update. We don't care that other people's records are updated while logging on. The chance of the two happening at the same time (Login + update for the same user) is minimal anyway.

Don't use this technique as a 'golden hammer' to solve all locking problems, in some cases you might want to block as data you are waiting for is being updated.

Other tips for avoiding deadlocks:

1. Always access tables in the same order in all your stored procedures/triggers consistently. This helps in avoiding deadlocks.
2. Keep your transactions as short as possible.
3. Touch as little data as possible during a transaction.
4. Never, ever, wait for user input in the middle of a transaction.
5. Make your front-end applications deadlock-intelligent, that is, these applications should be able to resubmit the transaction in case the previous transaction fails with error #1205.
6. In your applications, process all the results returned by SQL Server immediately, so that the locks on the processed rows are released, hence no blocking.

As always, consult BOL for more information.

### 2.7.10 Reading & writing configuration settings

As described in 2.2.13 with the following addition:

```
...
DECLARE @value varchar(25)
EXEC master.dbo.xp_regread @rootkey='HKEY_LOCAL_MACHINE',
    @key='Software\Demachina\Perfy\' ,
    @value_name='LogFile', @value=@value OUTPUT
```



```
PRINT @value  
...
```

If you are developing for SQL2000 exclusively you might want to create a user defined function for this purpose.

More information about reading and writing the registry from T-SQL can be found at [http://www.swynk.com/friends/green/xp\\_reg.asp](http://www.swynk.com/friends/green/xp_reg.asp).

#### 2.7.11 Pre process data

Although generally it is recommended to normalise data in a database, sometimes it is better from a performance and scalability point of view to de-normalise parts of the data that is frequently accessed. For example take the following scenario:

To display an i-Label (provider of detailed product details) icon whenever a product contains extended information we either need to perform a join between the product and the i-Label table or we need to resolve a relationship from the main Product object to the i-Label object. Both solutions use a lot of database resources (especially the latter). A simple optimisation can be made by batch processing the data in advance during the import of the data from the backend system. A simple flag on the Product table can then tell us if i-Label data is available for a product or not.

#### 2.7.12 Miscellaneous

Listed below are miscellaneous recommendations that do not fit in any of the other paragraphs:

1. Do not use 'SELECT \*' in your queries. Always write the required column names after the SELECT statement, like 'SELECT CustomerID, CustomerFirstName, City'. This technique results in less disk IO and less network traffic and hence better performance. It will also cause your code to be less error prone when inserting the results of a SELECT statement into a table.
2. Try to avoid server side cursors as much as possible. Always stick to a 'set based approach' instead of a 'procedural approach' for accessing / manipulating data. Cursors can be easily avoided by SELECT statements in many cases.
3. Avoid the creation of temporary tables while processing data as much as possible, as creating a temporary table means more disk IO. Consider advanced SQL, views, SQL2000 table variables or derived tables instead of temporary tables. Keep in mind that, in some cases, using a temporary table performs better than a highly complicated query.
4. Try to avoid wildcard characters at the beginning of a word while searching using the LIKE keyword, as that results in an index scan, which is defeating the purpose of having an index. The following statement results in an index scan, while the second statement results in an index seek:

```
SELECT LocationID FROM Locations WHERE Specialities LIKE '%pples'  
SELECT LocationID FROM Locations WHERE Specialities LIKE 'A%s'
```

Also avoid searching with not equals operators (<> and NOT) as they result in table and index scans. If you must do heavy text-based searches, consider using the Full-Text search feature of SQL Server for better performance.

5. Use 'Derived tables' wherever possible, as they perform better. Consider the following query to find the second highest salary from Employees table:

```
SELECT MIN(Salary)
FROM Employee
WHERE EmpID IN
(
    SELECT TOP 2 EmpID
    FROM Employee
    ORDER BY Salary DESC
)
```

The same query can be re-written using a derived table as shown below, and it performs twice as fast as the above query:

```
SELECT MIN(Salary)
FROM
(
    SELECT TOP 2 Salary
    FROM Employee
    ORDER BY Salary DESC
) AS a
```

This is just an example, the results might differ in different scenarios depending upon the database design, indexes, volume of data etc. So, test all the possible ways a query could be written and go with the efficient one. With some practice and understanding of how the SQL Server optimizer works, you will be able to come up with the best possible queries without this trial and error method.

6. While designing your database, design it keeping performance in mind. You can't really tune performance later, when your database is in production, as it involves rebuilding tables/indexes and re-writing queries. Use the graphical execution plan in Query Analyzer or SHOWPLAN\_TEXT or SHOWPLAN\_ALL commands to analyze your queries. Make sure your queries perform 'Index seeks' instead of 'Index scans' or 'Table scans'. A table scan or an index scan is a very bad thing and should be avoided where possible (sometimes when the table is too small or when the whole table needs to be processed, the optimizer will choose a table or index scan).
7. If you have a choice, do not store binary files, image files (Binary large objects or BLOBs) etc. inside the database. Instead store the path to the binary/image file in the database and use that as a pointer to the actual binary file. Retrieving and manipulating these large binary files is better

performed outside the database and after all, a database is not meant for storing files.

8. Minimize the usage of NULLs, as they often confuse the front-end applications. Any expression that deals with NULL results in a NULL output. ISNULL and COALESCE functions are helpful in dealing with NULL values.
9. In line with tip #1, always use a column list in your INSERT statements. This helps in avoiding problems when the table structure changes (like adding a column).
10. Though we survived the Y2K, always store 4 digit years in dates (especially, when using char or int datatype columns), instead of 2 digit years to avoid any confusion and problems. This is not a problem with datetime columns, as the century is stored even if you specify a 2 digit year. But it's always a good practice to specify 4 digit years even with datetime datatype columns.
11. Make sure each table has a primary key, preferably on a single column containing a numeric value.
12. Have Stored Procedures return primary keys using identity columns (in case of SQL server by passing back @@IDENTITY)?
13. Properly index your database (indexes on columns that are being searched, joined or ordered on)? Fields that are used for joins should preferably be of a numeric type. Prevent the use of varchar to perform joins.
14. Use the right datatype. Don't use varhars to store product ids if you have no leading zeros '001932'

## 2.8 Oracle PL/SQL

PL/SQL is the language used by Oracle's database server for programming stored procedures.

This chapter frequently refers to the ‘Oracle PL/SQL best practices’ book by O’Reilly, which is a must read for every PL/SQL developer.

### 2.8.1 Copyright message & page header

Listed below is an example of an actual PL/SQL page header. Every Package or stand alone procedure (See 2.8.11) needs to have one.

```

/*****
<Solution Name>

Copyright 2005, Demachina
www.demachina.com

All rights reserved

Document author: Jeroen Ritmeijer

VERSION      : 1.0
PACKAGE      : Customer_pkg

```

```
PURPOSE      : Contains all stored procedures and functions for Customer
               related functionality.
               This package contains the following procedures:
               - SelectCustomerByLoyaltyCard
DEPENDENCIES: RetailDatabase Tables
TO-DO        : Everything
KNOWN ISSUES: -
TAB SIZE     : 1
HISTORY      : 10/01/2004 - JR: Created initial version
***** /
```

## 2.8.2 Documenting stored procedures & functions

As described in 2.1.2. and 2.1.3 with the following addition:

A stand alone stored procedure contains only one function and therefore the function header needs to be added to the page header.

Each procedure header in a package header or body (See 2.8.11) is required to have a function header as displayed below.

```
/*-----
PROCEDURE : SelectCustomerByLoyaltyCard
PURPOSE   : As used by functional specifications chapter (1.1)
             This procedure returns the status of a PSA customer based on
             his customer card number. If the customer does not exist an
             empty cursor is returned
AFFECTS   : -
PARAMETERS: CustomerAccountID_IN    - Customer's account code
             cur_OUT                 - The resultset
RETURNS   : -
-----*/
```

*Stored procedure header*

## 2.8.3 Formatting Code

As described in 2.4.4 with the following additions:

Break up long statements and distribute them over multiple lines as displayed in the example below.

```
...
SELECT Modules.*, UserModules.UserID
FROM Modules WITH(NOLOCK)
LEFT JOIN UserModules WITH(NOLOCK, FASTFIRSTROW) ON
    Modules.ModuleID = UserModules.ModuleID AND
    UserModules.UserID = @UserID
ORDER BY Modules.Name
...
```

As can be seen in the two examples above, the convention is to capitalise all PL/SQL reserved keywords.

For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" will lead to confusion.

For more information see *Chapter 2 – Coding Style and conventions* of the before mentioned *PL/SQL Best practices book*.

#### 2.8.4 Naming conventions

Do not use spaces within the name of database objects, as spaces confuse front-end data access tools and applications.

Make sure to not use any reserved words for naming database objects, as that can lead to some unpredictable situations.

##### 2.8.4.1 Tables

Tables represent the instances of an entity. For example, you store all your customer information in a table. Here, 'customer' is an entity and all the rows in the customers table represent the instances of the entity 'customer'. So, name your table using the entity it represents, 'Customer' (singular, not plural).

Examples of valid table names are:

- 'Customer' for your customer table.
- 'CustomerOrder' for your order table (Don't use 'order' as this is a reserved keyword)
- 'ErrorMessage' for your error messages table.

This is a more natural way of naming tables, when compared to approaches which name tables as tblCustomers or tbl\_Orders. Further, when you look at your queries it's very obvious that a particular name refers to a table, as table names are always preceded by the FROM clause of the SELECT statement.

If your database deals with different logical functions and you want to group your tables according to the logical group they belong to, prefix your table name with a two or three character prefix that can identify the group.

For example, your database has tables which store information about Sales and Human resource departments, you could name all your tables related to Sales department as shown below:

- SL\_NewLead
- SL\_Territory
- SL\_TerritoryManager

You could name all your tables related to Human resources department as shown below:

- HR\_Candidate
- HR\_PremierInstitute
- HR\_InterviewSchedule

This kind of naming convention makes sure all the related tables are grouped together when you list all your tables in alphabetical order. However, if your database deals with only one logical group of tables, don't use this naming convention.

##### 2.8.4.2 Columns

Columns are attributes of an entity, that is, columns describe the properties of an entity. So, let the column names be meaningful and natural.

Examples of valid Names are:

- ID
- FirstName
- Address

As you can see, we do not use underscores in column names and capitalize the first character of each word.

#### 2.8.4.3 Stored procedures

Stored procedures always do some work for you, they are action oriented. So, let their name describe the work they perform and use a verb to describe the work.

Based on the type of function they perform, prefix the name with the action type (Select, Insert, Update or Delete). Note that a stored procedure does not always fit in one of these action categories, in that case do not prefix the stored procedure and just give it a logical name.

Examples of valid stored procedure names:

- DeleteUser
- SelectUsers
- ValidateUserForModule

Examples of invalid stored procedure names:

- GetUsers (use SelectUsers in stead)
- Validate\_User\_For\_Module (Don't use underscores)

Group Stored Procedures into packages depending upon the logical group they belong to. For more information see 2.8.11.

#### 2.8.4.4 Stored procedure parameter names

To avoid confusion when the parameter names of a stored procedure match field names of a SQL statement that is embedded in the procedure, suffix the parameter name with the parameter direction such as '\_IN' and '\_OUT'.

Confused? The example below will clarify things.

```
...
// ** HOW NOT TO DO IT **
Procedure SelectStore(StoreID IN VARCHAR2, cur_OUT OUT t_cursor)
IS
    v_cursor t_cursor;
BEGIN

    OPEN v_cursor FOR
        SELECT name
        FROM Store
        WHERE StoreID = StoreID;
...

```

StoreID = StoreID ????? That is not very clear, is it?

A proper, more readable, version is provided below

```
...
// ** A BETTER WAY TO DO IT **
Procedure SelectStore(StoreID_IN IN VARCHAR2, cur_OUT OUT t_cursor)
IS
    v_cursor t_cursor;
BEGIN
    OPEN v_cursor FOR
        SELECT name
        FROM Store
        WHERE StoreID = StoreID_IN;
...

```

#### 2.8.4.5 Indexes

Although you usually don't reference named indexes directly in your PL/SQL code, it is good practice to give them a useful name.

In stead of accepting the default, usually auto generated index names, our convention is the following:

IX\_<table name>\_<field name>

A possible Index name for the Customer.ID field is:

IX\_Customer\_ID

#### 2.8.5 Use Named arguments in complex SP calls

A simple trick to make your code more readable is to use named arguments when calling stored procedures with a less than obvious parameter list.

```
...
    LogErrorToTable
        ErrorNumber => 1004,
        SeverityLevel => 2,
        FriendlyDescription => ErrorString,
        FullDescription => sql,
        Source => 'LoadOrders/@Status=2032/UpdateOrderStatus=2',
        ErrorGroup => @DefaultErrorGroup
...

```

Only use it when it makes sense, don't use it when calling:

SelectProductsByStoreID 1

It is pretty obvious that '1' represents the StoreID in this call.

#### 2.8.6 Minimizing Code

Stored procedures should be used for data storage and retrieval only. This has a number of advantages:

1. **Scalability:** Databases are very difficult to scale beyond one machine and are therefore the main bottleneck in modern e-commerce applications. By minimising the complexity in a stored procedure the CPU is offloaded. The actual business logic should go into components (COM / .NET / Java) that are easy to distribute over multiple machines.



2. **Portability:** Less code means less chance of using database specific and proprietary code. This makes it easier to port the database to a different platform such as SQL Server, DB2 or Informix.
3. **Simplicity:** Stored procedures are difficult to debug. Less code means less chance of bugs.

#### 2.8.7 Error handling

Unlike SQL Server, Oracle's PL/SQL offers proper structured exception handling using the EXCEPTION section.

A primer on exception handling is out of the scope of this document, for specific tips with regards to this subject see *Chapter 5 – Exception handling* of the before mentioned *PL/SQL Best practices book*.

Error handling goes hand in hand with transactions, for more information see 2.8.8.

#### 2.8.8 Transactions

Transactions can be a lot more complicated than people think. Not everything is done for you automatically and you have to be really careful about what you are doing.

A discussion of transactions out of the scope of this document. The important thing is to remember they exist and to use them when necessary.

#### 2.8.9 Take care of locking

Locks are a natural thing in database environments as every experienced database programmer will tell you. Make sure your application detects locking problems and recovers gracefully.

A full discussion of locking is out of the scope of this document, but some tips for avoiding deadlocks are provided below.:

1. Keep your transactions as short as possible.
2. Touch as little data as possible during a transaction.
3. Never, ever wait for user input in the middle of a transaction.
4. Make your front-end applications deadlock-intelligent, that is, these applications should be able to resubmit the transaction in case the previous transaction fails.
5. In your application, process all the results returned by the database server immediately, so that the locks on the processed rows are released, hence no blocking.

#### 2.8.10 Pre process data

Although generally it is recommended to normalise data in a database, sometimes it is better from a performance and scalability point of view to de-normalise parts of the data which is frequently accessed. For example take the following scenario:



To display an i-Label (provider of complex product data) icon whenever a product contains extended information we either need to perform a join between the product and the i-Label table or we need to resolve a relationship from the main Product object to the i-Label object. Both solutions use a lot of database resources (especially the latter). A simple optimisation can be made by batch processing the data in advance during the import of the data from the backend system. A simple flag on the Product table can then tell us if i-Label data is available for a product or not.

### 2.8.11 Use packages in stead of stand alone stored procedures

Oracle has the option to group stored procedures together into so called packages. This is much easier to maintain than endless lists of ungrouped and stand alone stored procedures.

The so called package header contains the definitions of all package variables, types and procedures. The actual implementation is provided in the package body.

```
CREATE OR REPLACE PACKAGE "RETAILDATABASE"."PRODUCT_PKG" as

...
// add copyright header here
...

-- REF CURSOR, used for returning records to ADO.NET
TYPE t_cursor IS REF CURSOR;

/*-----
PROCEDURE : SelectProductsByName
PURPOSE   : Return products which partially match the specified name.
AFFECTS   : -
PARAMETERS: ProductName - The product name to look search for, if this var
                           contains 'null' the all records are returned.
                           cur_OUT      - The cursor containing the matching records.
RETURNS   : -
-----*/
Procedure SelectProductsByName(ProductName_IN IN VARCHAR2,
                               cur_OUT OUT t_cursor);

/*-----
PROCEDURE : InsertProduct
PURPOSE   : Insert a single product and return its primary key.
AFFECTS   : Products table
PARAMETERS: ProductName_IN - The name of the product to insert
                           Product_ID_OUT - return value contains the new records PK
RETURNS   : -
-----*/
Procedure InsertProduct(ProductName_IN IN Products.Name%TYPE,
                       Product_ID_OUT OUT Products.Product_ID%TYPE);

...

end PRODUCT_PKG;
```

Typical Oracle package header

An example of the package body is provided below:

```
CREATE OR REPLACE PACKAGE BODY "TEMPLATE"."PRODUCTS_PKG" as

/*-----
FUNCTION   : SelectProductsByName
PURPOSE   : Return products which partially match the specified name.
AFFECTS   : -
-----*/
```

```

PARAMETERS: ProductName - The product name to look search for, if this var
                        contains 'null' the all records are returned.
                        cur_OUT      - The cursor containing the matching records.
RETURNS      : -
-----*/
Procedure SelectProductsByName(ProductName IN VARCHAR2,
                               cur_OUT OUT t_cursor)
IS
    v_cursor t_cursor;
BEGIN

    IF ProductName is null
    THEN
        OPEN v_cursor FOR
            SELECT product_id, name
            FROM products;
    ELSE
        OPEN v_cursor FOR
            SELECT product_id, name
            FROM products
            WHERE name = ProductName;
    END IF;
    cur_OUT := v_cursor;
END SelectProductsByName;

/*-----*/
FUNCTION    : InsertProduct
PURPOSE     : Insert a single product and return its primary key.
AFFECTS     : Products table
PARAMETERS: ProductName_IN - The name of the product to insert
            Product_ID_OUT - return value contains the new records PK
RETURNS     : -
-----*/
Procedure InsertProduct(ProductName_IN IN Products.Name%TYPE,
                        Product_ID_OUT OUT Products.Product_ID%TYPE)
IS
BEGIN

    -- Get the new value for the primary key
    SELECT PRODUCT_ID_SEQ.NEXTVAL INTO Product_ID_OUT FROM DUAL;

    INSERT INTO Products
    (Product_ID, Name)
    VALUES
    (
        Product_ID_OUT,
        ProductName_IN
    );

END InsertProduct;

...

end PRODUCTS_PKG;

```

### 2.8.12 Use data dictionary types when declaring variables

Another useful Oracle feature is that you can refer to the type of a table column when declaring variables. This will save you from the hassle of updating all your stored procedures when the type of a field is changed.

```

...
Procedure InsertProduct(ProductName_IN IN Products.Name%TYPE,
                        Product_ID_OUT OUT Products.Product_ID%TYPE);
...

```

Use this feature whenever possible!!!!

### 2.8.13 Miscellaneous

Listed below are miscellaneous recommendations that do not fit in any of the other paragraphs:

1. Do not use `'SELECT *'` in your queries. Always write the required column names after the `SELECT` statement, like `'SELECT CustomerID, CustomerFirstName, City'`. This technique results in less disk IO and less network traffic and hence better performance. It will also cause your code to be less error prone when inserting the results of a `SELECT` statement into a table.
2. Try to avoid server side cursors as much as possible. Always stick to a 'set based approach' instead of a 'procedural approach' for accessing / manipulating data. Cursors can be easily avoided by `SELECT` statements in many cases.
3. Avoid the creation of temporary tables while processing data as much as possible, as creating a temporary table means more disk IO.
4. Try to avoid wildcard characters at the beginning of a word while searching using the `LIKE` keyword, as that results in an index scan, which is defeating the purpose of having an index. The following statement results in an index scan, while the second statement results in an index seek:

```
SELECT LocationID FROM Locations WHERE Specialities LIKE '%pples'  
SELECT LocationID FROM Locations WHERE Specialities LIKE 'A%s'
```

5. Use 'Derived tables' wherever possible, as they perform better
6. While designing your database, design it keeping performance in mind. You can't really tune performance later, when your database is in production, as it involves rebuilding tables/indexes and re-writing queries. Use the graphical execution plan, for example the one in Oracle's 'SQL Scratchpad' to analyze your queries. Make sure your queries perform 'Index seeks' instead of 'Index scans' or 'Table scans'. A table scan or an index scan is a very bad thing and should be avoided where possible (sometimes when the table is too small or when the whole table needs to be processed, the optimizer will choose a table or index scan).
7. If you have a choice, do not store binary files, image files (Binary large objects or BLOBs) etc. inside the database. Instead store the path to the binary/image file in the database and use that as a pointer to the actual binary file. Retrieving and manipulating these large binary files is better performed outside the database and after all, a database is not meant for storing files.
8. Minimize the usage of `NULLs`, as they often confuse the front-end applications. Any expression that deals with `NULL` results in a `NULL` output.
9. In line with tip #1, always use a column list in your `INSERT` statements. This helps in avoiding problems when the table structure changes (like adding a column).

10. Though we survived the Y2K, always store 4 digit years in dates (especially, when using char or int datatype columns), instead of 2 digit years to avoid any confusion and problems. This is not a problem with datetime columns, as the century is stored even if you specify a 2 digit year. But it's always a good practice to specify 4 digit years even with datetime datatype columns.
11. Make sure each table has a primary key, preferably on a single column containing a numeric value.
12. Have Stored Procedures return primary keys.
13. Properly index your database (indexes on columns that are being searched, joined or ordered on)? Fields that are used for joins should preferably be of a numeric type. Prevent the use of varchars to perform joins.
14. Use the right datatype. Don't use varchars to store product ids if you have no leading zeros '001932'

## 2.9 HTML

### 2.9.1 Copyright message & page header

We rarely deliver standalone HTML pages that don't contain any ASP code. However, in the unlikely event that we ever make one add the following header to the beginning of your page.

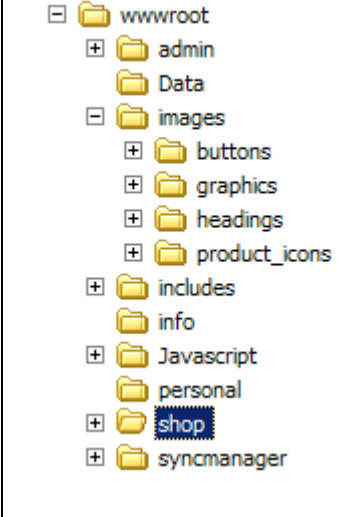
```
<!-- *****  
'                                     <Solution Name>  
'  
'                                     Copyright 2005, Demachina  
'                                     www.demachina.com  
'  
'                                     All rights reserved  
'  
'                                     Document author: Jeroen Ritmeijer  
'  
' FILENAME      : index.html  
' PURPOSE       : Display a static splash page.  
' MODULE        : Home page  
' DEPENDENCIES  : styles.css  
'  
' TO-DO         : -  
' KNOWN ISSUES  : -  
' TAB SIZE      : 4  
' HISTORY       : 25/02/2002 - JR - Created file  
***** -->
```

### 2.9.2 Formatting Code

Make sure the HTML is optimised and excess tags, that are usually added by visual HTML tools such as DreamWeaver, are removed.

### 2.9.3 Directory structure

As described in 2.1.16, don't throw everything in one directory, but categorise your HTML files. The same is true for graphics (buttons, images, headers and backgrounds). Listed below is an example taken from an actual project.

	<p>The most important directories are discussed below:</p> <ul style="list-style-type: none"> <li>• <b>Data:</b> all files that deal with data interaction between the server and client side JavaScript libraries.</li> <li>• <b>Images:</b> ALL images go into this folder or a subfolder of this folder. Do not place any images in a directory outside of the 'images' directory as this makes it difficult to zip all images up in one go.</li> <li>• <b>Includes:</b> Contains all ASP include files.</li> <li>• <b>JavaScript:</b> All client side JavaScript files are stored in this directory.</li> </ul>
---	---

Because every project is different there is no single recommended directory structure. The thing to remember is to structure it, but not to go overboard. Don't start to create directory structures that are 4 levels deep. The structure explained above is a good starting point.

### 2.9.4 Use style sheets

All modern pc based web browsers support style sheets. There are a number of reasons to use them:

1. **Maintenance:** Using style sheets, you can change the look and feel of an entire site by making a single change in one file.
2. **Readability:** HTML code is much better to read when every other tag is not a FONT tag.
3. **Compactness:** Because all formatting elements can be omitted the files are much more compact and therefore download faster.

One of the most important things to remember is to use proper names for you style elements.

For example, name an element MenuHeader rather then BlueBoldHeader as the style may change over time. A MenuHeader will still be a menu header, but it might be green and in italics.

Prevent the use of inline style definitions as much as possible, but rather create an external style sheet file. Try to generalize certain styles so they are applied by default, e.g.

<pre>A {</pre>	<pre>    FONT-FAMILY      : verdana;     FONT-SIZE        : 9px;     HEIGHT           : 17px;     BACKGROUND-COLOR : #C4D2F3; }</pre>
----------------	---

Using the above style definition every <A> tag will automatically be formatted using the specified style. This is much better than the following example:

```
.myFancyTXT {FONT-FAMILY: verdana; FONT-SIZE: 9px; HEIGHT: 17px;
BACKGROUND-COLOR: #C4D2F3;}
```

Firstly, this line is formatted in an unreadable fashion, additionally you can only apply it to the <A> tag by explicitly adding the style's name to the CLASS attribute. This an error prone process.

### 2.9.5 Form elements

Be careful when pre-populating elements with data. Make sure that it doesn't contain any invalid or reserved characters. Take the following as an example.

A customer's default delivery note is:

*Please leave the goods at the address with the nameplate "Johnson".*

If we pre-populate the delivery note on the order form with this line then we might cause an *invalid string termination*. For example:

```
<INPUT TYPE="TEXT" NAME="DeliveryNote" VALUE="Please leave the goods
at the address with the nameplate "Johnson"."">
```

The second quote character causes the string to be terminated after the word 'nameplate'. Therefore, always make sure you properly HTML encode the text before you use it as input for the 'Value' attribute. The HTML encoded version of the string would look like this:

*Please leave the goods at the address with the nameplate  
&quot;Johnson&quot;;*

### 2.9.6 Validate forms using JavaScript

Before sending off anything to the server, provide at least minimal validation in the browser. This will result in a more positive user experience as the user doesn't have to wait for a server roundtrip to see the errors. The best way to perform this validation is to make use of the ONSUBMIT event of the form element as described in the example below.

```
<html><head>
  <script language=javascript>
    function ValidateForm(oFrm)
    {
      if(oFrm.Address.value == "")
      {
        alert("Please fill in a valid address");
        return false;
      }
      else
        return true;
    }
  </script>
</head>
<body>
```

```
<form method="post" action="dosomething.asp"
      onsubmit="return ValidateForm(this)">
  <input type="text" name="Address">
  <input type="submit">
</form>

</body>
</html>
```

Note the following:

1. The function `ValidateForm` is called with 'this' as a parameter. 'this' in this case passes a reference to the current form. Doing it this way prevents awkward lookups such as 'document.forms.formname.element.value'.
2. `ValidateForm` is invoked from the `onsubmit` event. Doing it this way assures that the function is ALWAYS being called before the form is submitted, even when the enter key is pressed to submit the form.
3. Because we use 'return `ValidateForm(this)`' the value that is returned by the function either cancels (false) or proceeds (true) with the submit process.
4. We use an `<input type="submit">` element to submit a form. The only other valid alternative is `<input type="image">` .

DO NOT use a hyperlink or an `<input type="button">` element to submit a form as this requires unnecessary JavaScript functions to be added to the page.

### 2.9.7 Browser compatibility

Check your work in all required browsers. Minimise or eliminate the use of proprietary or non standard technologies such as layers or flash. These technologies make it much more difficult to maintain a site or add an additional channel such as interactive TV.

## 2.10 Java

This chapter outlines the conventions for Java. Industry standard conventions are used and are pretty much that specified by Sun that can be found at <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>.

### 2.10.1 Copyright message & page header

The page header is a little different from the standard header as Java source is documented into different sections.

At the very top of the source file insert the copyright notice comment as follows. This is not a Javadoc comment but a regular C style comment. Include dependencies and a list of known issues – keep these current.

```

/*****
 *
 *                               <Solution Name>
 *
 *                               Copyright 2005, Demachina
 *                               www.demachina.com
 *
 *                               All rights reserved
 *
 * FILENAME: Dummy.java
 * DEPENDENCIES:
 * KNOWN ISSUES:
 *
 *****/

```

Package and import statements then follow. Package names should be prefixed with com.<company name>.<project>. Imports should be fully qualified class names not wildcards. Eclipse and most decent IDEs have the ability to organise the import statements for you.

A class level javadoc comment should then follow that and begin with a description of the class and how it is used including code examples where appropriate. Also in this javadoc section put your name as an @author tag.

```

...
/**
 * The Example class provides ...
 * @author ewan.harrow
 */
public class Example { ...

```

### 2.10.2 Document the code

As described in 2.1.2 with the following addition:

It is standard in Java source to mark sections of code that require rework with a TODO comment. Eclipse and many IDEs will create a summary of these to-do comments. The example below illustrates typical use.

```

/**
 * Creates a string
 * @return the new test string
 */
public String stringTest() {
    //TODO: this needs refactoring so that these values are params
    String name = "address";
    String val = "7 Cowper Road";
    StringBuffer sb = new StringBuffer();
    sb.append("The value of ").append(name).append(" is ").append(val);
    return sb.toString();
}

```

### 2.10.3 Documenting methods

As described in 2.1.3 with the following changes. All public/protected methods and variables or attributes must be documented with full javadoc, which includes @param and @return tags – this includes getters and setters. For non obvious private methods it is advised that you add javadoc as well.

Use javadoc comments correctly. See <http://java.sun.com/j2se/javadoc/writingdoccomments/index.html> for the definitive guide.



```

/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 *           name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}

```

#### 2.10.4 Documenting the package

Each package i.e. source directory should include a package.html file containing text/html that describes what function the package performs with code examples where appropriate. This is used by the Javadoc tool to produce package level information.

#### 2.10.5 Formatting Code

As described in 2.1.5. Put the braces at the end of the line not on the line below.

#### 2.10.6 Naming conventions

There is defacto naming convention for Java defined by Sun. A full discussion is out of the scope of this document, but a summary is provided below.

Identifier	Case	Example
Class	Pascal	AppDomain
Exception class	Pascal	WebException Note Always ends with the suffix Exception.
Static field/Constant	C style const	RED_VALUE
Interface	Pascal	Runnable Note Do not begin the name with an I.
Method	Camel	toString
Package	Lower Case	com.eot.positionserver Note Use reverse domain name

Parameter	Camel	typeName
Property	Camel	backColour
Protected instance field	Camel	redValue

Static final variables are of course always in UPPER CASE as is the Java convention and where there are a number of variables which are variations of the same use a naming convention that is consistent and begin the name with the least significant common stem of the name. We do this as the javadoc tool and IDEs will list these in their lexicographic order i.e. alphabetically. For example, say we declare a couple of select SQL statements in the source, as they are both selects it would make sense to name them something containing the words select and SQL to distinguish them from delete or updates.

```
...
public static final String SELECT_SQL_ALL_ADDRESS = "SELECT * FROM ADDRESS";
public static final String SELECT_SQL_ALL_USERS = "SELECT * FROM USERS";
public static final String DELETE_SQL_ALL_ADDRESS = "DELETE FROM ADDRESS";
```

### 2.10.7 Object Orientation

Do not try functional programming in Java and to this end only use static methods where the operation does not involve an object's state. Never use non-final non-static public variables – properties or attributes should only be exposed via public get/set methods.

### 2.10.8 Error handling

- Use exceptions for exceptional circumstances.
- Javadoc exceptions thrown by a method.
- Do not ignore exceptions. Catching an exception and ignoring it is unacceptable. At the very least log it.
- Do practice exception translation. To prevent pollution of the higher level interfaces of an API do not allow exceptions from a lower level to be propagated up as it confuses users of the interface and risks coupling. Do include the original exception (exception chaining) where appropriate.
- Do add any state information to the exception message where appropriate. Top tip – if you override the toString() you can just append the object instance to the message.
- Do not throw generic exceptions
- For each new package create an abstract exception that other business exceptions in the package must extend. This adds more semantic meaning to the exceptions.

```
...
// Exception example with translation and chaining
try {
    // Use lower-level abstraction to do our bidding
```

```
...
} catch(LowerLevelException e) {
    throw new HigherLevelException("Problem when doing operation A on object B:
    " + B, e);
}
```

### 2.10.9 Concatenating strings

Keep in mind that concatenating many strings in a loop can become very slow due to constant memory re-allocation requests.

When performing a lot of concatenations, usually in a tight loop to turn a recordset into HTML or XML, don't use the '+' operator, instead use a StringBuffer.

Don't go overboard with the StringBuffer, it is perfectly acceptable to use the '+' operator when only a small number of string are concatenated together. It will make your code more readable as well.

For an example see 2.3.8 in the C# section..

### 2.10.10 Use of Reflection

Don't use it in areas where performance matters unless you absolutely have to. In addition minimise the use of the instanceof operator as it comes at a cost – use polymorphism instead.

### 2.10.11 Do override equals, toString and hashCode

Providing a good toString implementation makes your class much more pleasant to use. If you've provided a good toString method, generating a useful diagnostic message is as easy as:

```
...
System.out.println("Failed to connect: " + phoneNumber);
```

When practical, the toString method should return *all* of the interesting information contained in the object. For general use, particularly as a debugging aid, prefix the message with the class name followed by comma separated name value attribute pairs.

```
...
/**
 * @see java.lang.Object#toString()
 */
public String toString() {
    StringBuffer sb = new StringBuffer();
    sb.append("Dummy[address=").append(address).append(",
town=").append(town).append("]");
    return sb.toString();
}
```

Obey the general contract when overriding equals. Overriding the equals method seems simple, but there are many ways to get it wrong, and the consequences can be dire so do it but do it correctly.

Always override hashCode when you override equals. They are tied together and used by HashMap and Hashtable. Implement it incorrectly and you will

have bugs. Read the javadoc for hashCode and follow the contract and for a decent recipe read Item 8 in Effective Java by Joshua Bloch

<http://developer.java.sun.com/developer/Books/effectivejava/Chapter3.pdf>.

### 2.10.12 Logging

Log4j from the apache project has become the defacto standard for logging in Java applications even though an official logging subsystem and API has been added to the jdk1.4.

Declare the logger as static and final using the class name as the logger name as in the example below.

```
...
public class Dummy {
    private static final Logger log =
        Logger.getLogger(Dummy.class.getName());

    /** Creates a string
     * @return
     */
    public String stringTest() {
        log.debug("Attempting to create a test string");
        //TODO: these needs refactoring so that these values are params
        String name = "address";
        String val = "7 Cowper Road";

        StringBuffer sb = new StringBuffer();
        sb.append("The value of ").append(name).append(" is ").append(val);
        return sb.toString();
    }
}
```

Logging is invaluable throughout an application but excessive logging can create just too much noise so use logging levels to enable filtering. In principal use INFO for events such as starting and stopping of the app and major state changes, WARN for non-catastrophic recoverable events and ERROR for major problems that would require attention by a sysop. Debugging, the DEBUG level, should be used extensively but bear in mind that this is not very granular so non-useful debug such as “got here” is not helpful and just clogs up the logs.

Log exceptions as high up as you can if they are rethrown, converted or chained.

Add useful state information to aid problem solving. Top tip - if you override an object's toString() you can just append the object instance to the message.

### 2.10.13 Design & Patterns

Be familiar with software patterns. Most designs can be categorised into a well known software design pattern. By applying an appropriate well known pattern you not only solve the immediate problem but communicate the solution to those developers that you work with and follow you in the future.

Use them – don't invent your own.

See <http://developer.java.sun.com/developer/restricted/patterns/J2EEPatternsAtAGlance.html>

#### **2.10.14 Unit Testing**

- JUnit (<http://www.junit.org>) is used for unit testing.
- Write your unit test before you write the code. You will know you're your code works when the test passes.
- When fixing a bug write a unit test to reproduce the bug and you will know that you have fixed the bug when the test passes.
- Each class should have an associated test class in the same package. The naming convention is <classname>Test.java. Test classes should be included in the same directory as it is easy then to see which source lacks tests – when using Ant we can ensure that test classes are not packaged up for distribution so having them in the same source directory is not a problem.
- Ensure that your unit tests are unit tests and not module tests.

#### **2.10.15 Builds**

Use a proper build tool such as 'Ant'.

#### **2.10.16 Reading & writing configuration settings**

It is important to make configuration as idiot proof as much as possible for a deployer. In an ideal world we would use a cross platform means to configure an application which ideally would mean using `java.util.preferences` api but in practice it means using a mix of property and xml files as they are often easier for a deployer to manage.

XML files are preferred but in the absence of a standard schema property files may be used. To get around the issue of two config types we use the Jakarta Configuration sandbox library which acts as an adaptor/wrapper for the two and closely mirrors the `java.util.preferences` api.