Coding Standards

Netwin Info Solutions Pvt. Ltd

Version 1.0

For any comments, queries and suggestion mail at
Coding.Standards@netwininfoindia.com

Revision History

Date	Version	Description
28/02/2013	1.0	Initial Version

1. Introduction

It is the goal of this document to assist with writing good code. The following are the characteristics of good code.

a) Reliable
b) Maintainable
c) Efficient

Most developers are inclined towards writing code for higher performance, compromising reliability and

maintainability. But considering the long term ROI (Return On Investment), efficiency and performance comes below reliability and maintainability. If your code is not reliable and maintainable, you (and your company) will be spending lot of time to identify issues, trying to understand code etc throughout the life of your application.

a) Purpose:

To develop reliable and maintainable applications, you must follow coding standards and best practices.

The naming conventions, coding standards and best practices described in this document are compiled from various Microsoft and non Microsoft guidelines.

There are several standards that exist in the programming industry. You may follow any standard approach, but it is most important to ensure everyone is following the same standard.

b) Scope:

This document describes the coding practices and procedures recommended for all software development, in groups of 1 or more.

The guidelines are "lightweight" enough to be easy to follow, but complete enough to enable new developers to join a coding project quickly.

This document contains guidelines for writing web, desktop and mobile applications.

c) Reference Documents:

a. http://www.gemini.edu/documentation/webdocs/spe/spe-c-q0009.pdf

b. http://www.point64.com/code/dotnet/Csharp-coding-standards.pdf

c. http://physics.ucsd.edu/~emichels/Coding%20Guidelines.pdf

d) How to follow standards across the team

a. Make sure the coding standards are followed throughout the project development.

b. Each members needs to go through this document and make sure that each and every items are followed in their project.

c. Team leader is responsible to make sure that the team members are following the standards during implementation (strategy needs to be devised by the leader to have a timely check on the members if they are following the standards or not).

d. For a maintenance project, there is no need to rewrite standard code for the code that is already done instead make sure the standards are followed from now on.

2. Source Code Format

a. Naming Conventions

Pascal Casing - First character of all words are Upper Case and other characters are lower case.

Example: BlackColor

Camel Casing - First character of all words, except the first word are Upper Case and other characters are lower case.

Example: backColor

i. Variables and Constants

Variable: The **Variable** name is the usual way to reference the stored value. The identifier in computer source code can be bound to a value during run time, and the value of the variable may thus change during the course of program execution.

- In all Programming languages, Variables are declared using *Camel Casing*.
- Do not use variable names that resemble keywords. - While giving name to the variables do not use special keywords. E.g. Date, Class.

- Use Meaningful, descriptive words to name variables. Do not use abbreviations. E.g. **String address** not **String addr.**

- Avoid Long names <15 characters for variables
- Avoid names that are similar or differ only in case
- Instead of using _ in the name use capital letters e.g. FirstName,LastName.
- **Naming UI Elements:** While Naming UI Elements in any Programming language add UI Element prefix to the function description.
  Example: A Login Button can be given Names as btnLogin. Following Table gives an Idea of how to write UI Element Names.

Control	**Prefix**
Label	lbl
TextBox	txt
Button	btn
ImageButton	imb
Hyperlink	hlk
DropDownList	ddl
ListBox	lst
DataList	dtl
Repeater	rep
Checkbox	chk
CheckBoxList	cbl
RadioButton	rdo
RadioButtonList	rbl

Image	img
GroupBox	gbx
Panel	pnl
PlaceHolder	phd
Table	tbl
Validators	val
DataGrid	dg
DataGridView	dgv
GridView	gv
SQLDataSource	sds

Constants: A **Constant** is an identifier whose associated value cannot typically be altered by the program during its execution. Although a constant's value is specified only once, a constant may be referenced many times in a program.

- Give Constant Names with All Capital Letters.
- If there is a need of using Two words in defining a Constant then use an Underscore(_) between the words.
- Example: Tax rate constant can be defined as **int TAX_RATE**.

ii. Functions and Classes

Functions: A **Function** or **Sub Routine** is a sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed. Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.

- The name given to the Function should be descriptive which describe the purpose of the Function.
- The name should be related to the domain.

Classes: A C**lass** is a construct that is used to create instances of it – referred to as *class instances*, *class objects*, *instance objects* or simply *objects*. A class defines constituent members which enable its instances to have state and behavior.

Data field members (*member variables* or *instance variables*) enable a class instance to maintain state. Other kinds of members, especially *methods*, enable the behavior of class instances. Classes define the type of their instances.

A class usually represents a noun, such as a person, place or thing, or something nominalized. For

example, a "Banana" class would represent the properties and functionality of bananas in general. A single, particular banana would be an instance of the "Banana" class, an object of the type "Banana".

- In Any Object Oriented Programming Language, Class Names are written using Pascal Casing.
- Use whole words — avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

iii. File Names

Filenames need to be transparent, meaningful, user-friendly and consistent. This ensures that anyone who reviews a filename should immediately be able to form an accurate expectation about the content of the document.

- **Transparent** means that the words in the filename accurately summarize the content of the document using language in common use.
- **Meaningful** means that the words used in the filename usefully distinguish the content of this document from other documents.
- **User-friendly** means that the filename is easy to read and understand in relation to other filenames in a list, when it is presented in an on-screen window.
- **Consistent** means that similar principles of file naming are used by all document creators in a department, so that users do not have to interpret different conventions for the same collection of documents.

These principles, by the way, also apply to folder-naming conventions in a traditional folder structure – although with folders, because you are describing collections of content, you are more likely to need a standard taxonomy to provide you with standardized subject-related terms.

Below is the list of high level guidelines for how to approach establishing a common file naming convention:

- Reduce repetition of common words in file and folder names to a minimum
- Use acronyms only if they are commonly understood among users
- Keep filenames to below 20 characters if you can and File name should match with class name
- Start the filename with whatever is the key discriminating information for that document type when people want to go find it (e.g. date, correspondent)
- Construct filenames so that your IT system will display them in a helpful order
- Make sure that all newcomers to your workgroup are introduced to the file naming conventions
- When you begin any new folder structure or project, agree your file naming conventions among all content producers in advance
- Enforce consistency by periodic correction of inconsistent naming practices

File Names will follow Pascal coding in ASP.Net, mobile Apps and for PHP we will use all small case characters and words separated by '-'.

b. Comments and Documentation

Comments: A **comment** is a programming language construct used to embed programmer-readable annotations in the source code of a computer program. Those annotations are potentially significant to programmers but typically ignorable to compilers and interpreters. Comments are usually added with the purpose of making the source code easier to understand. The syntax and rules for comments vary and are usually defined in a programming language specification

- Program documentation is for programmers, not end users.

- While working in a Team, you have to follow the below points
- Other Programmers need to know how your code works
- Use *internal comments* for these programmers
- Tell them what your methods do, how to call them, and what they return
- Javadoc is the best way to document your code for users

Document as though, the next time you see your program, you will have forgotten everything about it!

a. Tags in Comments:

Certain tags are used in comments to assist in indexing common issues. Such tags are commonly syntax-highlighted within text editors and can be searched for with common programming tools, such as the Unix grep utility. Examples of tag conventions include:

- **FIXME** to mark potential problematic code that requires special attention and/or review.
- **NOTE** to document inner workings of code and indicate potential pitfalls.
- **TODO** to indicate planned enhancements.
- **XXX** to warn other programmers of problematic or misguiding code.

There is a risk that tags accumulate over time; it is advisable to include the date and the tag owner in the tag comment to ease tracking. Commenting Source code is very helpful while generating the project documentation at the time of Product delivery. It is good programming practice to write comments along with the source code.

The following Table gives a general idea on how to write comments in OOPs Languages.

All code documentation should have only initials of Programmer and not full name with Date in dd-mmm-yyyy format.

Comment Type	Usage	Example
Documentation Starts with /** and ends with */	Used before declarations of interfaces, classes, member functions, and fields to document them.	/** * Customer – a person or * organization */
C style Starts with /* and ends with */	Used to document out lines of code that is no longer applicable. It is helpful in debugging.	/* This code was commented out by NT */
Single line Starts with // and go until the end of the line	Used internally within member functions to document business logic, sections of code, and declarations of temporary variables.	// If the amount is greater // than 10 multiply by 100

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.
Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

Documentation: Software documentation or source code documentation is written text that accompanies computer software. It either explains how it operates or how to use it, or may mean

different things to people in different roles.

Role of documentation in software development

Documentation is an important part of software engineering. Types of documentation include:

- Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what shall be or has been implemented.
- Architecture/Design - Overview of software. Includes relations to an environment and construction principles to be used in design of software components.
- Technical - Documentation of code, algorithms, interfaces, and APIs.
- End User - Manuals for the end-user, system administrators and support staff.
- Marketing - How to market the product and analysis of the market demand.

All programming pages shall have a header stating about the purpose, date of creation, author initials, and dependencies like functions or procedures called inside this program.

```
/*
 * Programmer Name:VG
 * Purpose:Displaying News scroller
 * Date:28 Feb 2013
 * Dependancy: Home.aspx
 */
```

All functions, procedures will have purpose, date of creation, input & output parameters.

```
/*
 * Purpose: Calculate Discount
 * Date:28 Feb 2013
 * Input Parameter:
 *            int nDiscount – Discount value
 *            double nAmount – Amount on  which discount will be applied
 *
 *  Output Parameter:
 *            Double nUpdatedAmount – Amount after discount.
 */
```

All inside code modifications will have initials of programmer, date and purpose of change.

c. Statement formatting

i. **Indentation:** Indentation is used to format program source code to improve readability. Indentation is generally only of use to programmers; compilers and interpreters rarely cared how much whitespace is present in between programming statements.

- It is always a good programming practice to write only 80 characters in one Code line and also not more than 2000 lines of code in one file.
- It is suggested to maintain 4 Spaces or tab when starting a new block of Code.
- One blank line should be left between the Methods or Functions inside the class.

ii. **Spaces:** To keep code clear, feel free to use white space between operators, braces, and parenthesis. Code that is too dense is often difficult to read.

```
// Leave white space between operators
error = op_set_privileges();
// Use additional white space as appropriate for readability
if ( cmd > max_cmd )
{
// Let the operator know that the command wasn't executed
op_notify_error( NOT_EXECUTED ) ;
}
```

iii. **Braces:** Paired braces *should* always line up in the same column and the column is in alignment with the controlling statement**.** This makes it obvious where a block of code begins and ends. This is especially true for code that has several levels of nesting.

```
while (!done)
{
   if (idx == 5)
   {
     ...write the code;
   }
}
```

3. Exception Handling

a. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore non significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.

b. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.

c. Always catch only the specific exception, not generic exception.

d. Good:

```
void ReadFromFile ( string fileName )
        {
                try
                {
                        // read from file.
                }
                catch (FileIOException ex)
                {
                        // log error.
                        //  re-throw exception depending on your case.
                        throw;
                }
        }
```

Not Good:

```
void ReadFromFile ( string fileName )
{
  try
  {
    // read from file.
  }
  catch (Exception ex)
  {
    // Catching general exception is bad... we will never know whether
    // it was a file error or some other error.
    // Here you are hiding an exception.
    // In this case no one will ever know that an exception happened.
```

```
    return "";
  }
}
```

e. No need to catch the general exception in all your methods. Leave it open and let the application crash. This will help you find most of the errors during development cycle. You can have an application level (thread level) error handler where you can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application, or allowing the user to 'ignore and proceed'.

f. When you throw an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.

g. Good:

```
catch
{
        // do whatever you want to handle the exception

        throw;
}
```

h. Not Good:

```
catch (Exception ex)
{
        // do whatever you want to handle the exception

        throw ex;
}
```

i. Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means. For example, if you want to insert a record if it does not already exists in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur. On the other hand, you should always use exception handlers while you communicate with external systems like network, hardware devices etc. Such systems are subject to failure anytime and error checking is not usually reliable. In those cases, you should use exception handlers and try to recover from error.

j. Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will help you find which piece of code generated the exception and you can give specific error message to the user.

k. Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.

l. If opening database connections, always close them in the finally block.(Connection Pooling)

```
Sqlconnection con;
try
{
    //code to instantiate con object and provide connectionstring
    con.Open();
}
catch(Exception ex)
{
    //Exception handling code.
}
finally
{
    con.Close();
}
```

4. Database

a) Common Rules: -

- Do not let your front-end applications query/manipulate the data directly using SELECT or INSERT/UPDATE/DELETE statements. **Instead, create stored procedures, and let your applications access these stored procedures. This keeps the data access clean and consistent across all the modules of your application,** at the same time centralizing the business logic within the database.

- **Make sure you normalize your data at least till 3rd normal form. At the same time, do not compromise on query performance. A little bit of de-normalization helps queries perform faster.**

- **Write comments in your stored procedures, triggers and SQL batches generously, whenever something is not very obvious. This helps other programmers understand your code clearly.** Don't worry about the length of the comments, as it won't impact the performance, unlike interpreted languages like ASP 2.0.

- **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 fewer disks IO and less network traffic and hence better performance.

- **Views are generally used to show specific data to specific users based on their interest.** Views are also used to restrict access to the base tables by granting permission on only views. Yet another significant use of views is that, they simplify your queries. Incorporate your frequently required complicated joins and calculations into a view, so that you don't have to repeat those joins/calculations in all your queries, instead just select from the view.

- **Use 'User Defined Datatypes', if a particular column repeats in a lot of your tables, so that the datatype of that column is consistent across all your tables.**
  e.g. Suppose PhoneNo column is present in the Admin,Client,Users tables then care should be taken such that data type of the PhoneNo column in the all tables should be same.

- **Try not to use text, ntext datatypes for storing large textual data. 'text' datatype has some inherent problems associated with it. So, if you don't have to store more than 8 KB of text, use char(8000) or varchar(8000)datatypes.**

- 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, manipulating these large binary files is better performed outside the database and after all, database is not meant for storing files.

- **Use Unicode datatypes like nchar, nvarchar, ntext, if your database is going to store not just plain English characters, but a variety of characters used all over the world.** Use these datatypes, only when they are absolutely needed as they need twice as much space as non-unicode datatypes.

- **Always use a column list in your INSERT statements.** This helps in avoiding problems when the table structure changes (like adding a column). Here's an example which shows INSERT statement with a column list as shown below: INSERT INTO EuropeanCountries (CountryID, CountryName)VALUES (1, 'England')

- **To make SQL Statements more readable, start each clause on a new line and indent when needed. Also, reserved words should be uppercase to increase readability.**

  Following is an example:
  SELECT title_id, title
  FROM titles
  WHERE title LIKE 'Computing%' AND title LIKE 'Gardening%'

- As is true with any other programming language, **do not use GOTO. Excessive usage of GOTO can lead to hard-to-read-and-understand code.**

b) MySql: -

- Engine is always MyISAM except in special case when required different. All databases must use UTF8 charset.

- Database name must have the same name as project name. If the project name is "My Web Site" database should be created as:
  CREATE DATABASE mywebsite_db DEFAULT CHARSET UTF8;

- All tables must be UTF8 encoded. All tables in the same package must have prefix. Use 2 or 3 letters that describe the package to prefix database. If the project name is Real Deal Marketing, the most obvious prefix will be "rdm_";
  CREATE TABLE rdm_affiliates;

- ALTER TABLE and file versions major.minor[.build[.revision]]
  All alterations for all tables should be saved in a (database_name).sql file. If database name is "realdealmarketing" filename should be realdealmarketing-1.0.0.sql
  - Initial file must be version 1.0.0
  - If there are only **table alterations** file should have new version increased by 0.0.1
    Example: realdealmarketing-1.0.7.sql
  - If **new tables are created**, version must be increased by 0.1.

Example: realdealmarketing-1.2.0.sql

- If **tables are dropped**, version must be increased by 1.0. Example: realdealmarketing-2.0.0.sql
- All **minor revisions** should have appropriate file name and version following this pattern: major.minor[.build[.revision]]
- All versions should be saved in the same directory.

- Field names must be prefixed with 2-4 letters of table name.
  Field names are always lowercase with "_" to separate words
  All field names must be descriptive, avoid names like "unique", "sort" and reserved words.
  Example: -

```
DROP TABLE IF EXISTS rdm_affiliates;
CREATE TABLE IF NOT EXISTS rdm_affiliates
(
 aff_id              INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
 aff_url                  VARCHAR(120) NOT NULL DEFAULT '',
 aff_title                VARCHAR(120) NOT NULL DEFAULT '',
 aff_website     CHAR(10) NOT NULL DEFAULT '',
 aff_gender               CHAR(8) NOT NULL DEFAULT '',
 aff_landing_page        VARCHAR(120) NOT NULL DEFAULT '',
 aff_link                 VARCHAR(120) NOT NULL DEFAULT '',
 aff_text                 TEXT,

 PRIMARY KEY (aff_id),
 INDEX (aff_website)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8;
```

- **NULL and DEFAULT**: - All fields except AUTO_INCREAMENT, TEXT, DATE (and similar) must be defined as **NOT NULL DEFAULT** 'value'
  Examples: -

```
aff_title                VARCHAR(120) NOT NULL DEFAULT ''
aff_gender               ENUM('male','female') NOT NULL DEFAULT 'male'
user_id                  INT UNSIGNED NOT NULL DEFAULT 0
user_birthday   DATE DEFAULT NULL
user_allow_newsletter          TINYINT(1) UNSIGNED NOT NULL DEFAULT 0
art_text                 TEXT
```

- **Indices: -** All fields in WHERE and ORDER BY should be defined as INDEX. If you have query like this:
  `SELECT * FROM table1 WHERE a='something' ORDER BY b`
  Then fields a, b should be considered as indices
  `INDEX index_name (a,b)`

- **Char indices: -**
  If text field (CHAR, VARCHAR) is used as index for larger tables (> 5000 records) should be always defined as CHAR. For example, if you have fields like this:

```
aff_url VARCHAR(255) NOT NULL DEFAULT ''
...
INDEX afiliate_url (aff_url)
```

  To improve performance, this table should be changed to something like this

```
aff_url CHAR(20) NOT NULL DEFAULT ''
...
INDEX afiliate_url (aff_url)
```

- **Escape: -** Always escape values in WHERE even when value is integer

- **PHP Variables in Queries: -** PHP variables must be defined and escaped before query.
  Avoid: -
  ```
  $q = "SELECT * FROM users WHERE email='"._escape($_POST['email'])."' ";
  ```
  Use: -
  ```
  $email = _escape($_POST['email']);
  if (empty($email))
  {
          return false;
  }
  $q = "SELECT * FROM users WHERE email='{$email}' ";
  ```

- **PHPMyAdmin: -** Try to avoid PHP My Admin for table creation because it has problems with default values. It's great product for browsing and simple database manipulation but it makes you lazy and you usually forget to keep history of table alteration and other changes.

c) MS SQL Server: -

- **Naming conventions:** Give meaningful names to tables,columns,views,stored procedures
  Table: Customers
  Columns: CustomerID , CustomerFirstName , CustomerLastName
  Stored Procedure: GetCustomerDetails , InsertCustomerInfo Primary Key: pk_Customers_CustomerID
  Views: CustomersAddresses
- The name of the procedure should describe the purpose of the procedure. e.g SelectCustomesrInfo – To retrieve data from the Customer table.
  InsertCustomerInfo - To insert data to the Customer table. UpdateCustomerInfo - To update data of the Customer table. DeleteCustomerInfo - To delete data to the Customer table.
- Try **to avoid server side cursors as much as possible.** Always stick to 'set based approach' instead of a 'procedural approach' for accessing/manipulating data. Cursors can be easily avoided by SELECT statements in many cases. **If a cursor is unavoidable, use a simple WHILE loop instead, to loop through the table.** I personally tested and concluded that a **WHILE loop is faster than a cursor most of the times.** But for a WHILE loop to replace a cursor you need a column (primary key or unique key) to identify each row uniquely and I personally believe every table must have a primary or unique key.

- **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 or views or table variables of SQL Server 2000 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.e.g.
  ```
  DECLARE @TempEmployee TABLE (EmployeeID int, EmployeeName varchar(100),Primary Key(EmployeeID))
  OR
  CREATE TABLE @TempEmployee (EmployeeID int, EmployeeName varchar(100),Primary Key(EmployeeID))
  ```

- **Use the graphical execution plan in Query Analyzer or SHOWPLAN_TEXT or SHOWPLAN_ALL commands to analyze your queries.**Many times we use sub queries or joins in the queries which takes more time to execute. We can see this execution time required to fetch record from each table. For this go to the menu bar, select menu "Query" and inside that select sub menu "Include Actual Execution Plan".

- **Prefix the table names with owner names, as this improves readability,** avoids any

unnecessary confusions. Microsoft SQL Server Books Online even states that qualifying tables names, with owner names helps in execution plan reuse.

- **Use SET NOCOUNT ON at the beginning of your SQL batches,** stored procedures and triggers in production environments, as this suppresses messages like '(1 row(s) affected)' after executing INSERT, UPDATE, DELETE and SELECT statements. **This in turn improves the performance of the stored procedures by reducing the network traffic.**

- **Do not prefix your stored procedure names with 'sp_'. The prefix sp_ is reserved for system stored procedure that ship with SQL Server.**

- **Use char data type for a column, only when the column is non-nullable. If a char column is nullable, it is treated as a fixed length column in SQL Server 7.0+. So, a char(100), when NULL, will eat up 100 bytes, resulting in space wastage. So, use varchar(100) in this situation. Of course, variable length columns do have a very little processing overhead over fixed length columns. Carefully choose between char and varchar depending up on the length of the data you are going to store.**

- Minimize the usage of NULLs, as they often confuse the front-end applications, unless the applications are coded intelligently to eliminate NULLs or convert the NULLs into some other form. **ISNULL and COALESCE functions are helpful in dealing with NULL values.** Here's an example that explains the problem:Consider the following table, Customers which stores the names of the customers and the middle name can be NULL.
  CREATE TABLE Customers
  (
  FirstName varchar(20),
  MiddleName varchar(20),
  LastName varchar(20)
) Now insert a customer into the table whose name is Tony Blair, without a middle name:
  INSERT INTO Customers
  (FirstName, MiddleName, LastName)
  VALUES ('Tony',NULL,'Blair') The following SELECT statement returns NULL, instead of the customer name: SELECT FirstName + ' ' + MiddleName + ' ' + LastName FROM Customers
  To avoid this problem, use ISNULL as shown below: **SELECT FirstName + ' ' + ISNULL(MiddleName + ' ',") + LastName FROM Customers**

- **Always check the global variable @@ERROR immediately after executing a data manipulation statement (like INSERT/UPDATE/DELETE), so that you can rollback the transaction in case of an error (@@ERROR will be greater than 0 in case of an error).** e.g.

```
CREATE PROCEDURE addTitle(@title_id VARCHAR(6), @au_id VARCHAR(11), @title VARCHAR(20), @title_type CHAR(12))
AS
@title VARCHAR(20), @title_type CHAR(12))
 BEGIN TRAN
   INSERT titles(title_id, title, type)
    VALUES (@title_id, @title, @title_type)
   IF (@@ERROR <> 0)
   BEGIN
      PRINT 'Unexpected error occurred!'
      ROLLBACK TRAN
      RETURN 1
   END
 COMMIT TRAN
RETURN 0
```

- Though T-SQL has no concept of constants (like the ones in C language), variables will serve the same purpose. **Using variables instead of constant values within your SQL statements, improves readability and maintainability of your code.** Consider the following example:

  ```
  UPDATE dbo.Orders

  SET OrderStatus = 5
  ```

  The same update statement can be re-written in a more readable form as shown below:

  ```
  DECLARE @ORDER_PENDING int

  SET @ORDER_PENDING = 5

  UPDATE dbo.Orders

  SET OrderStatus = @ORDER_PENDING
  ```

d) Sqlite: -

- **Naming conventions:** Give meaningful names to tables,columns,views,stored procedures
  Table: Customers
  Columns: CustomerID , CustomerFirstName , CustomerLastName
  Stored Procedure: GetCustomerDetails , InsertCustomerInfo
  Primary Key: pk_Customers_CustomerID
  SQLite assigns a rowid column to every table you create. The rowid is a unique ID for each row. Never rely on rowid, particularly because older versions of SQLite would change the rowid values on tables that had deleted rows when you VACUUMed a database.
  So always give tables own artificial key and call it simply ID. So for the above Team table, it would have its first column defined as:

  ```
  ID INTEGER PRIMARY KEY AUTOINCREMENT
  ```

  Columns are always named using mixed case, except for short acronyms (such as ID). So for example, add team name and coach name columns they would be called **Name** and **CoachName**. Never, ever user the underscore (_) in any of my names.

- **Indexing: -**

  For column indexes, add Index to the column name (or names). If to put an index on the Team.Name column, it index would be called **NameIndex**. If it's a unique index then it should be **UniqueIndex**.

5) SVN Source Control

a) Introduction:

For Source control we are using Tortoise SVN.You need to follow below standards for any project.

a. Always create Repository for any new project.

b. Mail to Hardware Person for creating Project Repository

c. Then Hardware person will give your Project Repository path.

d. Link your working folder with same

e. Before commiting files verify the changes so that no Unnecessary changes are migrated to SVN.

f. If multiple developers are working on same project then take latest update from SVN before start of work daily

b) Do's and Don't for Source Control:

a. Dont add Unnecessary files in svn such as temp, thumbs.db or .suo

b. Do checkin and comments for all Major & Stable versions.

c. Do checkin atleast 3 times a day.

d. Do Take a Manual Backup on Backupserver daily EOD.

e. Do Use Show log functionality for verifying changes done by other users.

f. Dont Release any application before checkin of code.

c) Database Version Control

Make a script file for Create statements for tables, Procedure and functions.Update that file whenever you make modifications to database so we can track its changes.

6) Project Folder structure

You need to maintain below Project structure

If project name is OneMynd

then Directory structure will be

OneMynd
|
|--Code files
|
|--Database
|
|--Images
|
|--CSS
|
|--JS
|
|--Documents

7) Deployment Guidelines:

a. Don't copy unnecessary files on server.

b. Don’t modify files online directly. Test first on local, once tested then upload online
Don’t create page versions like index, index1, index_1, index2 etc on server. Maintain local copy of your earlier code pages.

c. Always take backup of online file version by downloading it , before updating files online.

d. Try to keep sensitive data encrypted.

e. Don’t keep backup files online.

f. Don’t upload SVN files on server.

For any comments, queries and suggestion mail at
Coding.Standards@netwininfoindia.com