Team 2 Coding Standards

Purpose of this document:

This document describes a collection of standards, conventions and guidelines for writing code and developing database features that are easy to understand, to maintain, and easy to enhance.

Important features of this document:

Existing standards from the industry are used wherever possible. The reason behind each standard is explained so that developers can understand why they should follow it. These standards are based on proven software-engineering principles that lead to improved development productivity, greater maintainability, and greater scalability. Standards are provided in separate sections for Java, JavaScript, HTML, and MySQL.

Why coding standards are important:

Coding standards for software are important because they lead to greater consistency within code of all developers. Consistency leads to code that is easier to understand, which in turn results in a code, which is easier to develop and maintain. Code that is difficult to understand and maintain runs the risk of being scrapped and rewritten.

Section 1 Java:

1. Naming Convention
    a. Use full English descriptors that accurately describe the variable/field/class/interface. For example, use names like firstName, grandTotal, or CorporateCustomer.
    b. Use terminology applicable to the domain. If the users of the system refer to their clients as Customer, then use the term Customer for the class, not client.
    c. Use mixed case to make names readable.
    d. Use abbreviations sparingly, but if you do so then use then intelligently and document it. For example, to use a short form for the word “number”, choose one of nbr, no or num.
    e. Avoid long names (<15 characters is a good tradeoff)
    f. Avoid names that are similar or differ only in case.
2. Comments
    a. Comments should add to the clarity of code.
    b. Avoid decoration, i.e., do not use banner-like comments
    c. Document why something is being done, not just what.
    d. Comment conventions are shown in the table below:

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C style Starts with /* and ends with */	Used to document out lines of code that are no longer applicable. It is helpful in debugging.	/* This code was commented out by Ashish Sarin */
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

3. Standard Techniques
    a. Document the code (Already discussed above).
    b. Paragraph/Indent the code: Any code between the { and } should be properly indented. One tab should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) will vary. If possible set tabs for every4 spaces.
    c. Line Length: Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.
    d. Wrapping Lines: When an expression will not fit on a single line, break it according to these general principles:
        i. Break after a comma.
        ii. Break before an operator.
        iii. Prefer higher-level breaks to lower-level breaks.
        iv. Align the new line with the beginning of the expression at the same level on the previous line.
        v. If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.
        vi. Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.
            1. longName1 = longName2 * (longName3 + longName4 - longName5)

               + 4 * longname6; // PREFER
            2. longName1 = longName2 * (longName3 + longName4

               - longName5) + 4 * longname6; // AVOID
    e. Declaration:

i. One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

ii. Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

f. Use white space:

   i. Blank lines improve readability by setting off sections of code that are logically related.
      1. Two blank lines should always be used in the following circumstances:
         a. Between sections of a source file
         b. Between class and interface definitions
      2. One blank line should always be used in the following circumstances:
         a. Between methods
         b. Between the local variables in a method and its first statement
         c. Before a block or single-line comment
         d. Between logical sections inside a method to improve readability
   ii. Blank spaces should be used in the following circumstances:
      1. A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
    ...
}
```

      2. A blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.
      3. A blank space should appear after commas in argument lists.
      4. All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.
         a. Example:

```
a += c + d;
a = (a + b) / (c * d);
```

```
while (d++ = s++) {
   n++;
}
```

5. The expressions in a for statement should be separated by blank spaces. Example: for (expr1; expr2; expr3)
6. Casts should be followed by a blank space. Examples: myMethod((byte) aNum, (Object) x);

g. Specify the order of Operations: Use extra parenthesis to increase the readability of the code using AND and OR comparisons. This facilitates in identifying the exact order of operations in the code.

h. Write short, single command lines Code should do one operation per line So only one statement should be there per line.

4. Standards For Member Functions

a. Naming member functions: Member functions should be named using a full English description, using mixed case with the first letter of any non-initial word capitalized. The first word of the member function should be a verb. This results in member functions whose purpose can be determined just by looking at its name. Examples:

   i. openAccount()
   ii. printMailingList()
   iii. save()
   iv. delete()

b. Naming Accessor Member Functions:

   i. Getters: member functions that return the value of a field / attribute / property of an object.
      1. Use prefix "get" to the name of the field / attribute / property if the field in not Boolean. Example: getFirstName().
      2. Use prefix "is" or 'has' or 'can' to the name of the field / attribute / property if the field is Boolean. Example: isPersistent().
   ii. Setters: member functions that modify the values of a field. Use prefix 'set' to the name of the field. Example: setFirstName().
   iii. Constructors: member functions that perform any necessary initialization when an object is created. Constructors are always given the same name as their class. Examples: Customer(), SavingsAccount().

c. Member Function Visibility: A good design requires minimum coupling between the classes. The general rule is to be as restrictive as possible when setting the visibility of a member function. If member function doesn't have to be public then make it protected, and if it doesn't have to be protected than make it private.

d. Documenting Member Function Header: Member function documentation should include the following:

    i. What and why the member function does what it does
    ii. What member function must be passed as parameters
    iii. What a member function returns
    iv. Known bugs
    v. Any exception that a member function throws
    vi. Include a history of any code changes
    vii. Examples of how to invoke the member function if appropriate.
    viii. If a member function is overloaded or overridden it should also be documented.
    ix. Note: It's not necessary to document all the factors described above for each and every member function because not all factors are applicable to every member function.

5. Standards for Fields (Attributes / Properties)

    a. Naming Fields: Use a Full English Descriptor for Field Names Fields that are collections, such as arrays or vectors, should be given names that are plural to indicate that they represent multiple values. Examples: firstName, orderItems, sqlDatabase.
    b. Naming Components: Use full English descriptor postfixed by the widget type. This makes it easy for a developer to identify the purpose of the components as well as its type. Example: okButton, customerList, fileMenu, newFileMenuItem.
    c. Naming Constants: In Java, constants, values that do not change, are typically implemented as static final fields of classes. The convention is to use full English words, all in upper case, with underscores between the words. Example: MINIMUM_BALANCE, MAX_VALUE, DEFAULT_START_DATE.
    d. Field Visibility: Fields should not be declared public for reasons of encapsulation. All fields should be declared private and accessor methods should be used to access / modify the field value. This results in less coupling between classes as the protected / public / package access of field can result in direct access of the field from other classes

6. Standards for Local Variables

    a. Naming Local Variables: Use full English descriptors with the first letter of any non-initial word in uppercase.

        i. Naming Streams: When there is a single input and/or output stream being opened, used, and then closed within a member function the convention is to use in and out for the names of these streams, respectively.
        ii. Naming Loop Counters: i, j, k, etc. should be used as loop counters.
        iii. Naming Exception Objects: The use of letter e for a generic exception.

    b. Declaring and Documenting Local Variables:

i. Declare one local variable per line of code
ii. Document local variable with an endline comment
iii. Declare local variables immediately before their use
iv. Use local variable for one operation only. Whenever a local variable is used for more than one reason, it effectively decreases its cohesion, making it difficult to understand. It also increases the chances of introducing bugs into the code from unexpected side effects of previous values of a local variable from earlier in the code.

7. Standards for Statements
    a. Simple Statements: Each line should contain at most one statement. Example:

        i. argv++;   // Correct
        ii. argc--;   // Correct
        iii. argv++; argc--;   // AVOID!
    b. Compound Statements: Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". See the following sections for examples.
        i. The enclosed statements should be indented one more level than the compound statement.
        ii. The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
        iii. Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.
    c. return Statements: A return statement with a value should not use parentheses unless they make the return value more obvious in some way.
    d. if, if-else, if else-if else Statements: The if-else class of statements should have the following form:
        i.
```
if (condition) {
    statements;
}
```
        ii.
```
if (condition) {
    statements;
} else {
    statements;
}
```
        iii.
```
if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```
    e. for Statements: A for statement should have the following form:

i. for (initialization; condition; update) {

```
    statements;
}
```

ii. An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form: for (initialization; condition; update);

iii. When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

f. while Statements: A while statement should have the following form:

i. while (condition) {

```
    statements;
}
```

g. do-while Statements: A do-while statement should have the following form:

i. do {

```
    statements;
} while (condition);
```

h. switch Statements: A switch statement should have the following form:

i. switch (condition) {

```
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

ii. Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

iii. Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

i. try-catch Statements: A try-catch statement should have the following format:

    i. try {

    statements;

    } catch (ExceptionClass e) {

       statements;

    }

    ii. A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully:

    iii. try {

    statements;

    } catch (ExceptionClass e) {

    statements;

    } finally {

      statements;

    }

8. Standards for Classes

    a. Class Visibility: Use package visibility for classes internal to a component

    b. Use public visibility for the façade of components.

    c. Naming classes: Use full English descriptor starting with the first letter capitalized using mixed case for the rest of the name.

    d. Documenting a Class:

        i. The purpose of the class
        ii. Known bugs
        iii. The development/maintenance history of the class

    e. Ordering Member Functions and Fields. The order should be:

        i. Constructors
        ii. private fields
        iii. public member functions
        iv. protected member functions
        v. private member functions
        vi. finalize()

9. Standards for Interfaces

    a. Naming Interfaces: Name interfaces using mixed case with the first letter of each word capitalized. Prefix the letter “I” or “Ifc” to the interface name

    b. Documenting Interfaces:

        i. The Purpose
        ii. How it should and shouldn’t be used

10. Standards for Packages
    a. Local packages names begin with an identifier that is not all upper case
    b. Global package names begin with the reversed Internet domain name for the organization
    c. Package names should be singular
    d. Documenting a Package
        i. The rationale for the package
        ii. The classes in the packages

11. Standards for Compilation Unit (Source code file)

    a. Naming a Compilation Unit: A compilation unit should be given the name of the primary class or interface that is declared within it. Use the same name of the class for the file name, using the same case.

    b. Beginning Comments:

```
/**
* Classname
*
* Version information
*
* Date
*
* Copyright notice
*/
```

Section 2 JavaScript:

1. JavaScript Files
   a. JavaScript programs should be stored in and delivered as .js files.
   b. JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to pageweight with no opportunity for mitigation by caching and compression.
   c. <script src=filename.js> tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components. There is no need to use the language or type attributes. It is the server, not the script tag, that determines the MIME type.
2. Indentation
   a. See Java standards
3. Line Length
   a. See Java standards
4. Comments
   a. See Java standards
5. Variable Declarations
   a. All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals. Implied global variables should never be used. Use of global variables should be minimized.
   b. The var statement should be the first statement in the function body.
   c. It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order if possible.
   d.
```
var currentEntry, // currently selected table entry
    level,        // indentation level
    size;         // size of table
```
   e. JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.
6. Function Declarations
   a. All functions should be declared before they are used. Inner functions should follow the var statement. This helps make it clear what variables are included in its scope.

b. There should be no space between the name of a function and the ((left parenthesis) of its parameter list. There should be one space between the) (right parenthesis) and the { (left curly brace) that begins the statement body. The body itself is indented four spaces. The } (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
function outer(c, d) {
    var e = c * d;
    function inner(a, b) {
        return (e * a) + b;
    }
    return inner(0, 1);
}
```

c. This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

```
        function getElementsByClassName(className) {
            var results = [];
            walkTheDOM(document.body, function (node) {
                var array,                // array of class names
                    ncn = node.className; // the node's classname
    // If the node has a class name, then split it into a list of simple names.
    // If any of them match the requested name, then append the node to the list of
    results.
                if (ncn && ncn.split(' ').indexOf(className) >= 0) {
                                results.push(node);
                }
            });
            return results;
        }
```

d. If a function literal is anonymous, there should be one space between the word function and the ((left parenthesis). If the space is omited, then it can appear that the function's name is function, which is an incorrect reading.

```
div.onclick = function (e) {
    return false;
};
that = {
    method: function () {
        return this.datum;
    },
    datum: 0
};
```

e. Use of global functions should be minimized.
f. When a function is to be invoked immediately, the entire invocation expression should be wrapped in parens so that it is clear that the value being produced is the result of the function and not the function itself.

```
var collection = (function () {
    var keys = [], values = [];
    return {
        get: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                return values[at];
            }
        },
        set: function (key, value) {
            var at = keys.indexOf(key);
            if (at < 0) {
                at = keys.length;
            }
            keys[at] = key;
            values[at] = value;
        },
        remove: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                keys.splice(at, 1);
                values.splice(at, 1);
            }
        }
    };
}());
```

7. Names

    a. Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 .. 9), and _ (underbar). Avoid use of international characters because they may not read well or be understood everywhere. Do not use $ (dollar sign) or \ (backslash) in names.

    b. Do not use _ (underbar) as the first or last character of a name. It is sometimes intended to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide private members. Avoid conventions that demonstrate a lack of competence.

    c. Most variables and functions should start with a lower case letter.

    d. Constructor functions that must be used with the new prefix should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time warning if a required new is omitted. Bad things can happen if new is not used, so the capitalization convention is the only defense we have.

    e. Global variables should be in all caps. (JavaScript does not have macros or constants, so there isn't much point in using all caps to signify features that JavaScript doesn't have.)

8. Statements

    a. See Java standards
    b. Avoid use of the continue statement. It tends to obscure the control flow of the function.
    c. The with statement should not be used.

9. Whitespace

    a. See Java standards

10. Block Scope

    a. In JavaScript blocks do not have scope. Only functions have scope. Do not use blocks except as required by the compound statements.

Section 3 HTML:

1. Validation: All HTML pages should be verified against the W3C validator to ensure that the markup is well formed. This in and of itself is not directly indicative of good code, but it helps to weed out problems that are able to be tested via automation. It is no substitute for manual code review. (For other validators, see HTML Validation in the Codex.)
2. Self-closing Elements: All tags must be properly closed. For tags that can wrap nodes such as text or other elements, termination is a trivial enough task. For tags that are self-closing, the forward slash should have exactly one space preceding it:
    a. <br /> The W3C specifies that a single space should precede the self-closing slash (source).
3. Attributes and Tags: All tags and attributes must be written in lowercase. Additionally, attribute values should be lowercase when the purpose of the text therein is only to be interpreted by machines. For instances in which the data needs to be human readable, proper title capitalization should be followed.
    a. For machines: <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    b. For humans:: <a href="http://example.com/" title="Description Here">Example.com</a>
4. Quotes: According to the W3C specifications for XHTML, all attributes must have a value, and must use double- or single-quotes (source). The following are examples of proper and improper usage of quotes and attribute/value pairs.
    a. Correct: <input type="text" name="email" disabled="disabled" />
       <input type='text' name='email' disabled='disabled' />
    b. Incorrect: <input type=text name=email disabled>
    c. In HTML, attributes do not all have to have values, and attribute values do not always have to be quoted. While all of the examples above are valid HTML, failing to quote attributes can lead to security vulnerabilities. Always quote attributes.
5. Indentation:
    a. As with PHP, HTML indentation should always reflect logical structure. Use tabs and not spaces.
    b. When mixing PHP and HTML together, indent PHP blocks to match the surrounding HTML code. Closing PHP blocks should match the same indentation level as the opening block.
    c. Correct:

```
<?php if ( ! have_posts() ) : ?>
	<div id="post-1" class="post">
		<h1 class="entry-title">Not Found</h1>
```

```
        <div class="entry-content">
            <p>Apologies, but no results were found.</p>
            <?php get_search_form(); ?>
        </div>
    </div>
<?php endif; ?>
```

d. Incorrect:

```
<?php if ( ! have_posts() ) : ?>
        <div id="post-0" class="post error404 not-found">
            <h1 class="entry-title">Not Found</h1>
            <div class="entry-content">
            <p>Apologies, but no results were found.</p>
<?php get_search_form(); ?>
            </div>
        </div>
<?php endif; ?>
```

<u>Section 4 MySQL:</u>

1. Naming
    a. Tables:
        i. Rules: Pascal notation; end with an ‘s’
        ii. Examples: Products, Customers
        iii. Group related table names (1)
    b. Stored Procs:
        i. Rules: sp<App Name>_[<Group Name >_]<Action><table/logical instance>
        ii. Examples: spOrders_GetNewOrders, spProducts_UpdateProduct
    c. Triggers:
        i. Rules: TR_<TableName>_<action>
        ii. Examples: TR_Orders_UpdateProducts
        iii. Notes: The use of triggers is discouraged
    d. Indexes:
        i. Rules: IX_<TableName>_<columns separated by _>
        ii. Examples: IX_Products_ProductID
    e. Primary Keys:
        i. Rules: PK_<TableName>
        ii. Examples: PK_Products
    f. Foreign Keys:
        i. Rules: FK_<TableName1>_<TableName2>
        ii. Example: FK_Products_Orderss
    g. Defaults:
        i. Rules: DF_<TableName>_<ColumnName>
        ii. Example: DF_Products_Quantity
    h. Columns:
        i. If a column references another table’s column, name it <table name>ID
        ii. Example: The Customers table has an ID column. The Orders table should have a CustomerID column.

i. General Rules:
    i. Do not use spaces in the name of database objects
    ii. Do not use SQL keywords as the name of database objects. In cases where this is necessary, surround the object name with brackets, such as [Year]
    iii. Do not prefix stored procedures with ‘sp_’ (2)
    iv. Prefix table names with the owner name (3)

2. Structure
    a. Each table must have a primary key
    b. In most cases it should be an IDENTITY column named ID
    c. Normalize data to third normal form
    d. Do not compromise on performance to reach third normal form. Sometimes, a little denormalization results in better performance.
    e. Do not use TEXT as a data type; use the maximum allowed characters of VARCHAR instead
    f. In VARCHAR data columns, do not default to NULL; use an empty string instead
    g. Columns with default values should not allow NULLs
    h. As much as possible, create stored procedures on the same database as the main tables they will be accessing.
3. Formatting
    a. Use upper case for all SQL keywords
    b. SELECT, INSERT, UPDATE, WHERE, AND, OR, LIKE, etc.
    c. Indent code to improve readability
    d. Comment code blocks that are not easily understandable
        i. Use single-line comment markers(--)
        ii. Reserve multi-line comments (/*.. ..*/) for blocking out sections of code
    e. Use single quote characters to delimit strings.
        i. Nest single quotes to express a single quote or apostrophe within a string
        ii. For example, SET @sExample = 'SQL''s Authority'
    f. Use parentheses to increase readability
        i. WHERE (color=’red’ AND (size = 1 OR size = 2))
    g. Use BEGIN..END blocks only when multiple statements are present within a conditional code segment.

h. Use one blank line to separate code sections.

i. Use spaces so that expressions read like sentences.

    i. fillfactor = 25, not fillfactor=25

j. Format JOIN operations using indents

    i. Also, use ANSI Joins instead of old style joins (4)

k. Place SET statements before any executing code in the procedure.

4. Coding

    a. Optimize queries using the tools provided by SQL Server (5)

    b. Do not use SELECT *

    c. Return multiple result sets from one stored procedure to avoid trips from the application server to SQL server

    d. Avoid unnecessary use of temporary tables

        i. Use 'Derived tables' or CTE (Common Table Expressions) wherever possible, as they perform better (6)

    e. Avoid using <> as a comparison operator

        i. Use ID IN(1,3,4,5) instead of ID <> 2

    f. Use SET NOCOUNT ON at the beginning of stored procedures (7)

    g. Do not use cursors or application loops to do inserts (8)

        i. Instead, use INSERT INTO

    h. Fully qualify tables and column names in JOINs

    Fully qualify all stored procedure and table references in stored procedures.

    i. Do not define default values for parameters.

        i. If a default is needed, the front end will supply the value.

    j. Do not use the RECOMPILE option for stored procedures.

    k. Place all DECLARE statements before any other code in the procedure.

    l. Do not use column numbers in the ORDER BY clause.

    m. Do not use GOTO.

    n. Check the global variable @@ERROR immediately after executing a data manipulation statement (like INSERT/UPDATE/DELETE), so that you can rollback the transaction if an error occur

        i. Or use TRY/CATCH

    o. Do basic validations in the front-end itself during data entry

p. Off-load tasks, like string manipulations, concatenations, row numbering, case conversions, type conversions etc., to the front-end applications if these operations are going to consume more CPU cycles on the database server

q. Always use a column list in your INSERT statements.

    i. This helps avoid problems when the table structure changes (like adding or dropping a column).

r. Minimize the use of NULLs, as they often confuse front-end applications, unless the applications are coded intelligently to eliminate NULLs or convert the NULLs into some other form.

    i. Any expression that deals with NULL results in a NULL output.

    ii. The ISNULL and COALESCE functions are helpful in dealing with NULL values.

s. Do not use the identitycol or rowguidcol.

t. Avoid the use of cross joins, if possible.

u. When executing an UPDATE or DELETE statement, use the primary key in the WHERE condition, if possible. This reduces error possibilities.

v. Avoid using TEXT or NTEXT datatypes for storing large textual data. (9)

    i. Use the maximum allowed characters of VARCHAR instead

w. Avoid dynamic SQL statements as much as possible. (10)

x. Access tables in the same order in your stored procedures and triggers consistently. (11)

y. Do not call functions repeatedly within your stored procedures, triggers, functions and batches. (12)

z. Default constraints must be defined at the column level.

aa. Avoid wild-card characters at the beginning of a word while searching using the LIKE keyword, as these results in an index scan, which defeats the purpose of an index.

bb. Define all constraints, other than defaults, at the table level.

cc. When a result set is not needed, use syntax that does not return a result set. (13)

dd. Avoid rules, database level defaults that must be bound or user-defined data types. While these are legitimate database constructs, opt for constraints and column defaults to hold the database consistent for development and conversion coding.

ee. Constraints that apply to more than one column must be defined at the table level.

ff. Use the CHAR data type for a column only when the column is non-nullable. (14)

gg. Do not use white space in identifiers.

hh. The RETURN statement is meant for returning the execution status only, but not data.

5. Reference:

1) Group related table names:

```
Products_USA
Products_India
Products_Mexico
```

2) The prefix sp_ is reserved for system stored procedures that ship with SQL Server. Whenever SQLServer encounters a procedure name starting with sp_, it first tries to locate the procedure in the master database, then it looks for any qualifiers (database, owner) provided, then it tries dbo as the owner. Time spent locating the stored procedure can be saved by avoiding the "sp_" prefix.

3) This improves readability and avoids unnecessary confusion. Microsoft SQL Server Books Online states that qualifying table names with owner names helps in execution plan reuse, further boosting performance.

4)
False code:

```
SELECT *
FROM Table1, Table2
WHERE Table1.d = Table2.c
```

True code:

```
SELECT *
FROM Table1
INNER JOIN Table2 ON Table1.d = Table2.c
```

5) Use the graphical execution plan in Query Analyzer or SHOWPLAN_TEXT or SHOWPLAN_ALL commands to analyze your queries. Make sure your queries do an "Index seek" instead of an "Index scan" or a "Table scan." A table scan or an index scan is a highly undesirable and should be avoided where possible.

6) Consider the following query to find the second highest offer price from the Items table:

```
SELECT MAX(Price)
FROM Products
WHERE ID IN
(
SELECT TOP 2 ID
FROM Products
ORDER BY Price Desc
)
```

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

```
SELECT MAX(Price)
FROM
(
SELECT TOP 2 Price
FROM Products
ORDER BY Price DESC
)
```

7) This suppresses messages like '(1 row(s) affected)' after executing INSERT, UPDATE, DELETE andSELECT statements. Performance is improved due to the reduction of network traffic.

8) Try to avoid server side cursors as much as possible. Always stick to a 'set-based approach' instead of a 'procedural approach' for accessing and manipulating data. Cursors can often be avoided by using SELECT statements instead. If a cursor is unavoidable, use a WHILE loop instead. For a WHILE loop to replace a cursor, however, you need a column (primary key or unique key) to identify each row uniquely

9) You cannot directly write or update text data using the INSERT or UPDATE statements. Instead, you have to use special statements like READTEXT, WRITETEXT and UPDATETEXT. So, if you don't have to store more than 8KB of text, use the CHAR(8000) or VARCHAR(8000) datatype instead.

10) Dynamic SQL tends to be slower than static SQL, as SQL Server must generate an execution plan at runtime. IF and CASE statements come in handy to avoid dynamic SQL.

11) This helps to avoid deadlocks. Other things to keep in mind to avoid deadlocks are:

· Keep transactions as short as possible.

· Touch the minimum amount of data possible during a transaction.

· Never wait for user input in the middle of a transaction.

· Do not use higher level locking hints or restrictive isolation levels unless they are absolutely needed.

12) You might need the length of a string variable in many places of your procedure, but don't call the LEN function whenever it's needed. Instead, call the LEN function once and store the result in a variable for later use.

13) IF EXISTS (SELECT 1 FROM Products WHERE ID = 50)
Instead Of:
IF EXISTS (SELECT COUNT(ID) FROM Products WHERE ID = 50)

14) CHAR(100), when NULL, will consume 100 bytes, resulting in space wastage. Preferably, use VARCHAR(100) in this situation. Variable-length columns have very little processing overhead compared with fixed-length columns.