1: General rules

1-1 Apart from prefixes and spaces exactly the same name is used for the field name, caption (for the form control that maintains the data), report column header and code variable name. We call this the "Cradle to the Grave Naming Convention".

> We can trace the footsteps of St. Patrick almost from his cradle to his grave by the names of places called after him.
>
> ***E. Cobham Brewer 1810–1897. Dictionary of Phrase and Fable.***

Good:

Field name:	CustomerFirstName	(table name prefix)
Field caption:	First Name	(space added for clarity)
Report column header:	First Name	
Form caption:	First Name	
Code variable:	`strFirstName`	(data type prefix, no spaces)

Bad:

Field name:	CustFName
Field caption:	Christian Name
Report column header:	Given Name
Form caption:	Name
Code variable:	`strFNm`

Believe it or not the second example is, almost universally, what you will find in the real world of commercial applications. Badly informed managers often erroneously believe that hacking together low-quality applications will result in them being completed faster. A very important responsibility of your job as a development professional is to educate and inform so that this doesn't happen!

1-2 There must only be one exit in any sub or function.

Code that has multiple exits is confusing to read and causes problems with clean up code (such as closing record sets, de-referencing objects, destroying objects and ensuring that warnings are always re-enabled).

Good:

```
Function CodeToName( strCategory as string) as string

Dim strReturn as string
IF strCategory = "MAN" Then
    strReturn = "Manchester"
Elseif strCategory = "LHR" Then
    strReturn = "London Heathrow"
Else
    strReturn = "Unknown"
End If
    CodeToName = strReturn
End Function
(Error handling code not shown)
```

Bad:

```
Function CodeToName( strCategory as string) as string

IF strCategory = "MAN" Then
     CodeToName = "Manchester"
```

```
     Exit Function
Elseif strCategory = “LHR” Then
     CodeToName = “London Heathrow”
     Exit Function
Else
    CodeToName = “Unknown”
    Exit Function
End If
End Function
```

Be particularly careful not to violate the single exit rule in error handling code.

Good:

```
Private Sub txtFilmYearReleased_Exit(Cancel As Integer)
    On Error GoTo ErrorHandler
    Me.lblHelp.Caption = ""
CleanUpAndExit:
    Exit Sub
ErrorHandler:
    Call MsgBox("An error was encountered" & vbCrLf & _
        vbCrLf & _
        "Description: " & Err.Description & vbCrLf & _
        "Error Number: " & Err.Number, , "Error")
    Resume CleanUpAndExit
End Sub
```

Bad:

```
Private Sub txtFilmYearReleased_Exit(Cancel As Integer)
    On Error GoTo ErrorHandler
    Me.lblHelp.Caption = ""
    Exit Sub
ErrorHandler:
    Call MsgBox("An error was encountered" & vbCrLf & _
        vbCrLf & _
        "Description: " & Err.Description & vbCrLf & _
        "Error Number: " & Err.Number, , "Error")
End Sub
```

1-3 Use mixed case names, no underscores and capitalised first letter throughout for sub, function, variable, field, object and control names.

Good:

```
strCompanyPostCode
txtCompanyPostCode
GetCompanyPostCode()
```

Bad:

```
strCompany_Post_Code
txtcompanypostcode
GetCompanypostcode()
```

By never using underscores in your own sub and function names it will always be obvious which subs are event handlers.

By never using underscores in variable names it will always be easy to differentiate between variables and constants.

1-4 Name variables in the form Noun or Noun–Verb.

Because of our *Cradle to the Grave Naming Convention* just about every variable name that relates to a field will begin with a noun. This is because every entity (table name) in the system is usually a noun. For example:

```
strCustomerFirstName
strCustomerLastName
strCustomerPostCode
```

Boolean variables often include a verb.

Good: `blnCustomerIsValid, blnCustomerHasCar`
Bad: `blnIsValidCustomer, blnHasCarCustomer`

1-5 Name subs and functions as in the spoken word.

All function and sub names should read just like the spoken English word. This makes code less quirky and nearer to "real life".

Good:
```
AddCustomer
DeleteCustomer
CalculateMovingAverage
GetCustomerByPrimaryKey
```

Bad:
```
CustomerAdd
CustomerDelete
MovingAverageCalculate
CustomerGetByPrimaryKey
```

You should be aware that there is a contrary school of thought (to which we do not subscribe) that suggests that subs and functions should be named in the form of noun-verb. The advantage of this approach is that an alphabetic listing of all sub and function names will group all subs relevant to a specific entity (such as Customer) together.

1-6 Never use abbreviations for table, field—or any other—names.

> A good yardstick for choosing a name is to try to imagine that there is an extraordinary reward for two programmers if they can independently come up with the same program text for the same problem. Both programmers know the reward, but cannot otherwise communicate. Such an experiment would be futile, of course, for any sizable problem, but it is a neat goal. The reward of real life is that a program written by someone else, which is identical to what one's own program would have been, is extremely readable and modifiable.
>
> ***Dr. Charles Simonyi (formerly Microsoft's Chief Architect)***

Example: A table contains Customer data. One of the fields in the table contains the Customer's first name.

Good: `strCustomerFirstName`
Bad: `strCustFNm, strCstmFName`

If you were playing Charles Simonyi's game and you needed to guess which name another programmer had come up with for a variable containing information of data type string that contains a Customer's first name you'd probably win (with strCustomerFirstName) in the first example above but would probably lose with the second two.

A long time ago, when visual basic did n't have a compiler and memory was a precious commodity, there was a small performance gain to be had by abbreviating names. There isn't any more (and hasn't been for some years). So why do 21st century programmers still create cryptic, bug prone, and difficult-to-maintain code by abbreviating names? It is one of the great mysteries of life!

Some programmers argue that you don't have to type as much if you use abbreviations. You may save a few keystrokes when you write the code but you'll waste many hours later on when you have to analyse every sub in order to establish what your cryptically named variables actually contain.

1-7 Name tables, fields — and everything else — in the singular.

As mentioned in the previous rule, One of the goals of our naming convention is that programmers should be able to intuitively guess the correct name of any variable or function.

The use of plurals makes this goal more difficult. Consider a *GetCompanyType(lngCompanyType)* function that returns zero, one or more company records. Without this rule programmers would have to guess whether the correct function was *GetCompanyType()* or *GetCompanyTypes()*.

This problem is avoided by only ever using singular names for variables, tables, subroutines, directory (folder) names, web page names, file names and field names.

The single exception to this rule is in the naming of object collections. While we haven't covered creating you own object collections in this book you can see that Microsoft adhere to this standard within the Access object model. For example the Forms collection contains many Form objects and the Controls collection contains many Control objects.

About Table Naming

You should be aware that some database designers turn this convention upsiled down and make all table names plural. It is (vastly) preferable to enforce an "everything in the singular" convention as field names must prefixed with the table name (see *table and field naming rules*) and would look very confusing in the plural.

1-8 Do not use "Magic Numbers".

Never use numeric values as control variables. Instead, include a globally visible constant.

Good: `GetCompany( TSM_NORTH_WEST_AREA_ONLY )`
Bad: `GetCompany( 23 )`

1-9 Strongly type all function and sub parameters.

Good:
```
Function GetCustomer( lngCustomerID as Long )
```
Bad:
```
Function GetCustomer( lngCustomerID)
```

1-10 Declare all parameters ByVal unless there is a good reason to declare them ByRef.

Good:
```
Function GetCustomer( ByVal lngCustomerID as Long )
```
Bad:
```
Function GetCustomer( lngCustomerID)
Function GetCustomer( ByRef lngCustomerID)
```

Both of the two "bad" case sabove are functionally equivalent because *ByRef* is the default method of passing arguments.

Good reasons to pass parameters ByRef would include code that must be heavily optimised for speed and a low memory footprint or the requirement to return multiple values from a sub or function.

1-11 Do not use globally scoped variables.

The use of global variables violates the concept of encapsulation. Global variables also result in buggy code that is difficult to maintain and prevents code re-use as code becomes reliant upon a supporting infrastructure.

Note that this rule does not apply to global constants.

1-12 Do not use the addition operator for concatenating strings.

Use & for concatenating strings and + only for arithmetic operations.

1-13 Avoid *Exit For* and *Exit Do*

> If you use an Exit Do, people will think less of you.
>
> ***Button seen at programmer's convention – unknown author.***

Bailing out of loops can cause the same problem as having multiple exits in sub as it make s code less readable and more prone to bugs. Unless there is no reasonable alternative avoid doing this.

1-14 Use the Call statement when calling sub and functions.

Consider the following example:

Without the *Call* statement

```
strCustomerName = GetCustomerName( lngCustomerID)
DeleteCustomer lngCustomerID
MsgBox "You have successfully deleted customer: " & strCustomerName
```

With the *Call* statement

```
strCustomerName = GetCustomerName( lngCustomerID)
Call DeleteCustomer( lngCustomerID)
Call MsgBox( "you have successfully deleted customer: " & strCustomerName)
```

The second example read s more cleanly than the first because the three function calls use identical syntax (brackets de-line ate arguments). For this reason *Call* should be used in every case when calling functions, sub s and object methods.

1-15 Never rely upon the default properties of objects.

VBA has a very confusing "feature" in that every control has a default property...

```
txtCustomerFirstName.Value = "John"
and
txtCustomerFirstName = "John"
```

...are functionally equivalent because *Value* is the default property of a *TextBox* control.

The default property feature has (thankfully) been removed from the latest versions of stand-alone VB (VB.Net and VB 2005) showing that Microsoft also agree that it isn't the best feature in the world.

Never use default properties in your VBA code as they make the code less readable and more prone to error (as the actual property being manipulated must remain in the programmer's memory).

2: Table and Field naming

2-1 Never use prefixes for table names.

Access developers commonly prefix table names with *tbl*. We don't like this convention. The primary object in an Access database is the table so we prefer the *lack of* any prefix to identify this type of object.

Table name prefixes are also incompatible with the "Field names are always prefixed by the table name" rule.

2-2 Primary keys are named using the syntax: <table name> + <ID>

Example: Table named *Customer*
PrimaryKey: *CustomerID*

2-3 Foreign keys always have exactly the same name as the related primary key.

Example: The primary key *MediaID* in the *Media* table is also called *MediaID* when used as foreign key in the *Film* table.

2-4 If not obvious, the unit of measure is incorporated into the field name.

Example: A field is needed to indicate the length of a film. It is not clear whether the data will be expressed in hours or minutes.

Good: FilmLengthMinutes
Bad: FilmLength

2-5 The link table in a many-to-many relationship is always named with the names of the tables on either side of the many-to-many relationship.

Example: Two tables have a many-to-many relationship, the *Film* and *Actor* tables.

A link table named *FilmActor* is created to model the relationship.

2-6 Apart from Foreign Keys, field names are always prefixed by the table name.

This is one of those golden rules that (in conjunction with the *Cradle to the Grave Naming Convention*) will massively increase your productivity and the reliability of your code.

Because all table names in a database are unique, every field name in your database (apart from Primary and Foreign Keys) will also be unique if you prefix all field names with table names. This provide shuge benefits:

- Many reporting tools (including Access) have query designers that automatically create joins for fields of the same name. Because we have a rule that foreign and primary keys always have the same name, the query designer will get it right every single time. There will never be any spurious joins caused by occurrences of the same non-key field name in different tables.
- SQL queries become a lot simpler when this rule is in place as you will not have to explicitly qualify every field name with a reference to the related table.
- Code becomes more readable and less error-prone:

```
strCustomerFirstName = "James" 'I know this originated in the Customer Table
strEmployeeFirstName = "Peter" 'I know this originated in the Employee Table
strFirstName = "Paul" 'Which table did this come from?  I'll have to guess or
                      'trace the code.
```

Example: All field names in the Film table are prefixed with the word *Film* so have names such as *FilmTitle*, *FilmYearReleased* and *FilmReview*.

3: Field properties

3-1 Primary keys are always meaningless and have the data type: *AutoNumber*.

3-2 The default value of non-required foreign keys should be Null and not 0 (the Access default).

Since the data type of primary keys must always be *AutoNumber* it follows that the data type of foreign keys are always numeric. It will not be possible to add records to a child table if it has a foreign key (that form s part of a constrained relationship) with a value of 0.

3-3 At least one (non primary key) field must always be required.

This rule will prevent users from accidentally creating entirely blank records. While observing this rule avoid non-essential required fields, only making them so when business rules demand it.

4: Access object naming

4-1 Prefix objects with the following letters in lowercase:

Object	Prefix
Table	No prefix
Query	qry
Form	frm
Report	rpt
Macro	mcr
Module	mod

Example: *frmFilm*.

5: Form control naming

5-1 Use generally accepted three-letter standard prefixes for all form controls. Some of the more common prefixes are:

Prefix	Object Type	Example
cbo	Combo Box and Drop Down List Box	cboEnglish
chk	Checkbox	chkReadOnly
cmd	Command Button	cmdOK
ctr	Control (when specific type is unknown)	ctrCurrent
dat	Data control	datFilm
dir	Directory listbox	dirSource
dlg	Common Dialog Control	dlgFileOpen
frm	Form	frmEntry
fra	Frame	fraStyle
img	Image	imgIcon
lbl	Label	lblHelpMessage

lin	Line	linVertical
lst	ListBox	lstPolicyCodes
mnu	Menu	mnuFileOpen
opt	Option Button	optRed
ole	OLE Control	oleWorksheet
pic	Picture	picHotel
spn	Spin Control	spnPages
txt	TextBox	txtLastName
tmr	Timer	tmrAlarm

6: Variable naming

6-1 Module scoped variables must be prefixed with the letter *m* and globally scoped variables must be prefixed with the letter *g*.

Note that, despite this standard, it is nearly always poor programming practice to use any globally visible variables in an application (note that this rule does not apply to globally visible constants that are extremely useful).

```
mstrCustomerName          ' Module level
strCustomerName           ' Local to sub
gstrApplicationLanguage   ' Globally visible
```

6-2 Use generally accepted three-letter variable prefixes.

Microsoft publish recommended three-letter variable prefixes in their MSDN library and these are generally accepted by professional programmers.

Some of the more common prefixes are:

Data Type	Prefix
String	str
Long integer	lng
Boolean	bln
Currency	cur
Double	dbl
Variant	vnt (var also acceptable)
Date and Time	dat (dtm also acceptable)

The alternative prefixes for Variant and Date and Time above are so commonly used that both can be allowed as similes within code.

Early code was often written with one letter variable type prefixes (such as s for string). There's even examples in wizard generated code of two-letter variable prefixes (such as st for string) but neither style is often seen today.

7: Constants

7-1 Name constants using uppercase and underscores.

Programmers should always be aware of which variables are constants. Clearly identify them by always using the uppercase/underscore style for constant names (and only for constant names). It will then always be easy to differentiate between variables and constants in your code.

Good: `TSM_APPLICATION_STATUS`
Bad: `conApplicationStatus`

You'll often see code written using the "bad" convention above. Note that Microsoft do not observe this rule with their own constants such as *vbRed*.

7-2 Constants should always be named with a prefix that is unlikely to be used by any other third party code.

Microsoft use the prefix vb for all of their constants. You should use at least three letters, perhaps your own initials or the name of the application you are writing. As there are 27^3 combinations of three-letter prefixes it is unlikely (though not impossible) that your constant prefixes will coincide with those used by a third party library.

8: Variable declaration

8-1 The *Require Variable Declaration* option must always be switched on.

This can be done by selecting Tools➪Options from the code editor window. The *Option Explicit* command will then be included at the top of every new (but not existing) module.

8-2 All variables must be strongly typed.

This includes variables declared as parameters for subs and functions.

Good:
```
Dim strFirstName as string
Sub DeleteCustomer( lngCustomerID as Long)
```
Bad:
```
Dim strFirstName
Sub DeleteCustomer( lngCustomerID)
```

8-3 All variables must be declared and typed with a dedicated Dim statement.

It is possible to declare more than one variable within a single *Dim* statement but this practice can cause problems.

At first glance you may think that:
```
Dim strOne,strTwo as string
```

Was functionally equivalent to:
```
Dim strOne as String
Dim strTwo as String
```

The first example would, in fact, declare *strOne* as a Variant and only *strTwo* as a string.

Good:
```
Dim strFirstName as string
Dim strLastName as string
```

Bad:
```
Dim strFirstName, strLastName as string 'first variable is a variant
```

8-4 When declaring variables and constants include an in-line comment detailing their purpose.

Example:

```
dim lngFormMode as long          ' Can be TSM_FORM_INSERT or TSM_FORM_UPDATE
dim lngCompanyCount as long      ' Used to iterate through rsCcompanies
dim blnCompanyIsActive as boolean' Flags current credit status
```

9: Error handling

9-1 Error handling must be implemented in every sub and function *without exception*.

Programmers often argue that some code is so simple that it can never fail so does not need error handling. While this may be true in some cases there's nothing wrong with a catch-all approach. If absolutely every subroutine has error handling you cannot possibly confront your user with an unprofessional and confidence-sapping run time error. When you take this approach it is comforting to find that those bullet-proof subs that couldn't possible fail often do, but when they do the error is gracefully handled.

9-2 Whenever SetWarnings method are used in a function or sub a call to SetWarnings(True) must be included immediately before the single exit point (within the clean up code section).

If code branches to the ErrorHandler after the DoCmd SetWarnings(False) method call but before you re-enable standard Access warnings with a DoCmd SetWarnings(True) method call there will be no more standard warnings during the entire Access session. This could be potentially disasterous. Adding a precautionary SetWarnings(True) call within the clean up section eliminate sexposure to this problem.

10: Object destruction

10-1 Record sets that are opened must be explicitly closed.

10-2 All objects that are instantiated must be explicitly destroyed when no longer needed.

This has been a "hot topic" amongst VBA programmers and fiercely debated on the programming bulletin boards for years.

Some programmers argue that Access can automatically de-reference object variables when they go out of scope (in the same way that other types of variables do).

It is widely believed that memory leaks (a situation where a computer gradually grinds to a halt as the memory become sexhausted and then needs to be re-booted) are often caused by relying upon automatic object destruction.

It is instructive to examine the wizard-generated code for Access switchboards. The code within the *HandleButtonClick* function shows that Microsoft also find value in closing record sets and explicitly destroying objects when working with their own product.