Table of contents

Introducing AVR for .NET

This document introduces several of the unique and powerful features of AVR, AVR's RPG. Whether you're a traditional RPG programmer with little or no experience with other languages, or a veteran VB/C# programmer, this document provides an overview of several of AVR's language concepts and syntactical details.

IBM introduced RPG back in the late 60s. With numerous transformations over the years, its current capabilities eclipse its original intent of simply creating reports. Alas, today, as the flagship language of the iSeries, IBM's RPG's remains a syntactically convoluted and narrow language—even with its myriad changes over the years. It remains hampered by its 24x80 character mode interface and it does little to empower the iSeries to address modern application demands such as Web enablement, supporting rich user interfaces, and implementing service-oriented solutions. The iSeries comprises a variety of RPG dialects (including ILE RPG and RPG/400). For this document's purposes I'll generically group IBM's host-based RPGs as "green-screen RPG."

ASNA Visual RPG (AVR), on the other hand, provides a sophisticated, highly-evolved version of RPG. It continues to offer standby RPG idioms and constructs (such as externally described data structures, key lists, parameter lists, record-level operations and program calls). However, it also improves on old-fashioned RPG with a modern, expressive syntax, and many new facilities and powers. With AVR, you can build rich Windows applications, powerful browser-based apps (for both Internet and intranet deployment), as well as create underlying business logic and database access components for service-oriented applications (including publishing and subscribing to Web services). While AVR is the nickname for ASNA Visual RPG the product, the name AVR also refers to Visual RPG's RPG dialect.

This document takes you a quick tour of some of the features and facilities offered by AVR. It's not intended to provide all the details you need for everything it describes; rather, this document's purpose is to introduce you to some of the ways AVR supercharges RPG.

Raising the bar on RPG

AVR's RPG substantially raises the bar on old-time, green-screen RPG. Syntactically, AVR's RPG has been extended to use an indented, block structure. This enhanced syntax lets you write more expressive RPG, removes green-screen RPG's dogged insistence on specific column position, and more naturally lends itself to AVR for .NET's event-driven environment. While AVR doesn't at first look like green-screen RPG, when you learn the basics of its syntax and fully appreciate the power it offers, I think you'll agree that AVR does a good job of merging the best of the old RPG model with the best of what .NET has to offer.

In addition to its numerous syntactical enhancements, AVR brings RPG fully into the world of object-oriented programming. With AVR, you can gracefully implement the three hallmarks of object-oriented programming: encapsulation, polymorphism and specialization. Before we dig into OO detail (that's covered later), lets take a closer look at AVR's syntax and some of its general features.

Green-screen RPG required you to write your code to a very specific columnar scheme (owing back to its reliance on punched cards in the days of its youth). AVR eschews specific columns with a keyword syntax very similar to that used by the iSeries's Command Language (CL). If you're familiar with RPG and CL,

it might help to think of AVR as a blend of the best CL's syntax with the power and structure of RPG. (If you're familiar with OS/400's V5Rx free-format RPG calculation spec, you''ll notice a few syntactical similarities between it and AVR as well.)

For example, this snippet of green-screen RPG:

```
C                     Z-ADD     0         TOTBAL
C*
C* Read 10 customer records.
C      1              DO        10
C                     READ      CUST                  50
C      *IN50          ADD       CMBAL     TOTBAL
C                     ENDDO
```

could look like this in AVR:

```
TotalBalance = 0

// Read 10 customer records.
Do FromVal( 1 ) ToVal( 10 )
    Read Cust
    If ( NOT %EOF( Cust ) )
        TotalBalance = TotalBalance  + CMBAL
    EndIf
EndDo
```

This snippet shows several things about AVR's RPG:

- statements are keyworded like CL
- green-screen RPG's column dependence is eliminated
- statements can (and should!) be indented
- whitespace, empty lines, and case don't matter
- two slashes (//) indicate a comment
- RPG built-in functions (BIFS) are supported (you'll later learn that there are also powerful alternatives to BIFs)
- implicit simple variable assignments are allowed (instead of requiring explicit operations like ADD or Z-ADD)
- long identifier names are available
- case doesn't matter—tokens and operations can be expressed in uppercase, lowercase, or any combination thereof
- operations are always listed before the operation's operands

Another more subtle thing about the AVR snippet above is the phrase "could look like this." AVR supports a full complement of backwards-compatible RPG constructs. However, as you'll learn in this document, it also offers powerful alternatives to many long-standing green-screen RPG idioms.

Let's take a closer look at AVR's RPG.

AVR leaves RPG column-dependence behind

AVR's RPG uses CL-like keywords to remove the need for specifically placing code in specific columns. In fact, if you come from an RPG background, you'll notice that AVR was styled almost as though RPG and CL were merged into one language, with CL-like command interfaces provided over all RPG operations. This isn't by accident. AVR retains a great deal of RPG's idioms and personality, but it also needed a more rational, expressive syntax. Given most RPG programmer's experience with CL, blending the best of both worlds was a natural for AVR. AVR applies the "command interface" concept to virtually all RPG operations as well as RPG specification types such as F-Specs and D-Specs.

AVR uses CL-like rules to govern its use of keywords. Like CL, keywords are optional. For example, if you were to write this code

```
Chain File( Cust ) Key( CustKey )
```

or

```
Chain Cust Key( CustKey )
```

or

```
Chain Cust CustKey
```

any of the three lines would compile. AVR's positional rules for keywords would apply and the first value would be assumed to be the from value and the second would be assumed to be the to value. If you omit keywords, the values must be provided in the order in which the RPG operation expects them.

If you specify a given keyword, all keywords following the given keyword must be provided. For example, you could use:

```
Chain Cust Key( CustKey )
```

but you could not use:

```
Chain File( Cust ) CustKey
```

The occurrence of the first keyword tells the compiler to ignore positional use from there on. Search AVR's online help for Operation codes to see each operation's keywords.

From a styling perspective, the general rule is to omit the first keyword and include all others for clarity's sake. The additional clarity keywords provide is generally worth the extra typing they require. For example, consider these two lines of code

```
Chain Cust CustNo
```

and

```
Chain Cust Key( CustNo )
```

Semantically, these lines are identical—the compiler accepts either. But the second version is much easier to read. Although the general rule is to omit the first keyword and include all others, there are exceptions. Consider this AVR:

```
Do 1 10 x
```

and

```
Do 1 ToVal( 10 ) Index( x )
```

and

```
Do FromVal( 1 ) ToVal( 10 ) Index( x )
```

These lines show the AVR Do loop in action. As you can see, for maximum clarity you will probably want to use all the keywords for a few of the AVR opcodes.

Long-time RPG coders often react negatively to the suggestion that keywords be used to improve code readability. I've heard more than one beginning AVR coder say, "If I wanted to write something as wordy as COBOL, I'd write COBOL"! You may consider the keyworded version too verbose and to require too much typing. However, remember that it's much harder to *read* code than it is to *write* code. Extra care taken to write highly readable code will pay huge dividends over the life of a program.

Experienced RPG programmers will notice that in most cases AVR's keywords map directly to columns in corresponding green-screen RPG operations. For example, in the Do statement above, the FromVal keyword maps to RPG's factor 1; the ToVal maps to RPG's factor 2; and the Index keyword maps to RPG's result column.

With some RPG operations, such as the Do operation, the keywords directly convey the keyword's purpose—without regard for what column the Do's operands used to occupy (in other words, the Do's keywords FromVal and ToVal don't reference factor 1 or factor 2 in any way). In others, such as the Z-ADD operation, the keywords reflect a more direct relationship with green-screen RPG columns. For example, a keyworded Z-ADD to add the value of 8 to variable x could be written as

```
ZAdd F2( 8 ) Result( x )
```

where F2 is shorthand for Factor 2 and Result is shorthand for the result column. See the AVR help for each operation's exact keywords.

Although AVR doesn't have a dependence on green-screen RPG's columns, you will see in later that a good AVR coding style does indeed use columns to format code—but that's for readability purposes, not for the compiler's sake.

If you are a VB or C# programmer and all this talk about RPG's columns have you confused, don't worry about it. It's your good fortune to be able to learn RPG and not ever have to worry about columns and exactly what goes in each column!

AVR ignores whitespace, empty lines, and case

Not only does AVR remove columnar restrictions, it's also very flexible with regard whitespace, empty lines, and case. For example, you could write

```
MyVar = 9
```

or you could use:

```
MYVAR = 9
```

and AVR would know that both MyVar and MYVAR are referring to the same variable. Case insensitivity also applies to all other program identifiers (such as routine names, file names and array names). There are two additional important points to make about AVR's case insensitivity:

- Just because case doesn't matter to AVR doesn't mean it doesn't matter to you and other programmers. Strive for diligent consistency with your use of case in your AVR programs.
- Although AVR's language is case insensitive, there will be times when you'll need to pay close attention to case—especially when specifying the keys to collections. This will be covered in more detail later.

AVR is also very liberal about accepting whitespace. For example, you could write

```
Chain Cust Key(CUSTKEY)
```

or

```
Chain Cust Key( CUSTKEY )
```

Any additional whitespace is ignored by the AVR compiler. AVR also allows blank lines. Whitespace, or the lack of it, doesn't matter to the compiler; but it matters greatly to programmers trying to read code. Plenty of whitespace and consistent casing dramatically improve program clarity and readability.

Continuing lines

AVR doesn't use an end-of-line mark character (as C# or green-screen RPG's free-format syntax does with the semi-colon). Each AVR line ends simply with a carriage return. There may be times, for readability purposes, when you want to continue a line to one or more other lines. Consider this DclDiskFile:

```
DclDiskFile Cust Type(*Input) Org(*Indexed) Prefix(Cust_) File("Examples/CMastNewL2")...
```

You'll often see such a line written using AVR's line continuation syntax (which, again, borrows from CL):

```
DclDiskFile  Cust +
      Type( *Input ) +
      Org( *Indexed ) +
      Prefix( Cust_ ) +
      File( "Examples/CMastNewL2" ) ...
```

While the latter requires more lines, it is much more readable. Continued lines must end with the + sign. Continued lines inside expressions are often vexing at first. For example, don't let this line of code trip you up:

```
MyValue = SomethingReallyLong ++
          SomethingElseReallyLong
```

In this case, the first plus sign is part of the expression, the second is the line continuation character.

AVR comments

Anytime AVR encounters two contiguous slashes it assumes they denote a comment area. Used at the beginning of a line, two slashes indicate the entire line is a comment. Used anywhere else in a line indicates that from the slashes to the end of the line is a comment. For example, the snippet below has three comments. One is a full line comment, the other two are partial-line comments.

```
// Read 10 customer records.
Do FromVal( 1 ) ToVal( 10 )
    Read Cust                // Read one record.
    If ( NOT Cust.IsEOF )    // Record not found.
```

AVR also supports CL/C#-like /* */ streaming comments. For example, you could also do this:

```
/*
 | Read 10 customer records.
 */
Do FromVal( 1 ) ToVal( 10 )
    Read Cust                // Read one record.
    If ( NOT Cust.IsEOF )    // Record not found.
```

where anything between /* and */ is considered a comment by the AVR compiler.

AVR statements can (and should!) be indented

Given AVR's whitespace rules, it shouldn't come as a surprise that you can use any indention style you want. You could write this:

```
If ( NOT Cust.IsEOF )
    TotalBalance  = TotalBalance  + CMBAL
EndIf
```

or you could write this:

```
If ( NOT Cust.IsEOF )
TotalBalance=TotalBalance + CMBAL
EndIf
```

However, it is highly recommended that you adopt a rational indention scheme so that the layout of your source code convey the logic of that code. For example, you could eschew indention and write this:

```
Do FromVal( 1 ) ToVal( RecordsToReadBackwards )
ReadP Cust BOF( BegOfFile )
If ( BegOfFile )
Leave
Else
Write CustMem
EndIf
EndDo
```

Or you could do this:

```
Do FromVal( 1 ) ToVal( RecordsToReadBackwards )
    ReadP Cust BOF( BegOfFile )
    If ( BegOfFile )
        Leave
    Else
        Write CustMem
    EndIf
EndDo
```

Which stub would you rather try to make sense out of later? The better and more consistent you are with indenting your code, the easier your AVR code will be to read and maintain. Consistent, deliberate use of proper indention is a hallmark of a great AVR program.

New data types

AVR introduces several new data types that green-screen RPG either didn't have or weren't widely adopted by RPG programmers. This section introduces a few of AVR's data types.

- **Boolean*. The Boolean data type represents a true or false value. Under the covers, *True resolves to the numeric value 1 and *False resolves to the numeric value zero. AVR provides two keywords, *True and *False, to use for *Boolean testing.
- **Indicator.* AVR's *Indicator data type is for backwards compatibility with green-screen RPG's named indicators. *While* it is similar to the *Boolean data type, *Indicator resolves to either character value "1" or character value "0." *Boolean is the preferred data type for true/false values and should be used over *Indicator (this is especially true if you think you'll ever want to share your AVR assemblies with C# or VB programmers).
- **Integerx* (where *x* is 2, 4, or 8). These three integer types represent two-, four-, and eight-byte integer values. RPG programmers should remember that the 2, 4, and 8 refer to bytes, not character positions (as is the case with zoned or packed variables). Thus the *Integer4 data type has a maximum value of 2,147,483,657 (not 9999 as might be assumed by a green screen RPG programmer). Generally, you should use the *Integerx data type any time you need a whole number program variable.
- **String*. Strings are alphameric values of undeclared length. They are similar to green-screen RPG's character data type (known to AVR as **Char*) but their length is dynamic and is determined by the variable's content. A string's actual length is determined at runtime. Generally, you should use a *String data type any time you need a character program variable. Not only are *Strings more convenient than *Chars in that strings aren't constraint to s specific length, as you learn more about AVR for .NET you'll learn that there are a couple of things that *Strings can do that *Char types can't.

These data types (and all others actually) do a lot more for you than simply represent a value. As you'll soon see, in AVR for .NET, all variables are instances of objects and these objects bring events, methods, and properties to the party. We'll cover what this means for you in more detail later, but in the meantime, pay close attention as to how data types are used in AVR—you notice that interesting, subtle power lurks for virtually all of AVR's data types. Here's just one example to pique your interest. Consider having *Packed, 8, 0 database field named LOANDT that stores the date as a numeric value in the format yyyymmdd. You could easily format this value like this:

```
LoanDt.ToString( "0000-00-00" )
```

This code is using the *Packed field's ToString() method, with a custom numeric formatting string, to render the numeric value as a string with the format given. In this case, zeros mean non-suppressed zero numeric place holders. If LoanDT's value were 20070602 the output from the code above would be "2007-06-02." Don't agonize too much right now about ToString() and how it's actually formatting the numeric value; rather understand that because virtually all data elements in AVR for .NET are instances of objects, lots of interesting power lurks just below the surface.

Beyond simply adding these (and many other) new data types to your RPG kitbag, you'll soon see that AVR supports all old familiar green-screen RPG data types. And, as alluded to above, you'll see that old friends such as **Packed*, **Zoned*, and fixed-length **Char* character strings aren't just supported, but that AVR highly supercharges them.

Simple variable assignments

Although AVR supports legacy RPG assignment operations such as ADD, Z-ADD and MOVE (along with the myriad related assignment operations), the preferred AVR way to perform variable assignments is with a simple equal statement. For example, you could write this:

```
ZAdd F2( 8 ) Result( x )
```

or

```
ZAdd 8 x
```

However, the preferred way to perform this assignment is to simply use the = assignment operator:

```
x = 8
```

AVR's simple assignment operation also has a very interesting implication on the strongly-typed nature of .NET. AVR's equal sign assignment operator is, for the most part, the semantic equivalent to green-screen RPG's MOVE operation code. MOVE did double duty as both an assignment operator and, if needed (and possible) a conversion operator. For example, AVR assignment operation allows this code:

```
DclFld Rate         Type( *String )
DclFld NumericValue Type( *Packed ) Len( 8, 0 )

Rate = "45678"
NumericValue = Rate // String assigned to numeric value!
```

In the example above, AVR automatically converts the string value to a numeric value—no explicit conversion is required. (As with green-screen's RPG's MOVE, the code above would fail if the string value contained a non-numeric character.) To RPG programmers familar with green-screen RPG's MOVE operation, this automatic conversion should seem pretty natural. To VB and C# programmers, though, it's borderline heresy! By its nature, .NET is very type-specific and generally doesn't tolerate the assignment of one data type to another. AVR does indeed follow nearly all the rules for type-specificity that C# and VB (with Option Strict = True) do. As you can see, though, AVR's equal assignment operation is more forgiving than what VB or C# allow. For the sake of VB and C# sharp programmers who are especially irked by this violation of type-specifity, relax. You could also write the above code pretty much like you would in VB or C# (In the code below Convert.ToInt32() is an intrinsic .NET numeric conversion method):

```
DclFld Rate         Type( *String )
DclFld NumericValue Type( *Packed ) Len( 8, 0 )

Rate = "45678"
NumericValue = Convert.ToInt32( Rate )
```

Do note that AVR's implicit assignment conversion only works with the *String data type. It does not work with the *Char data type. The following code would not compile

```
DclFld Rate         Type( *Char )
DclFld NumericValue Type( *Packed ) Len( 8, 0 )

Rate = "45678"
NumericValue = Rate
```

The attempted assignment of a *Char data type to a *Packed data type would cause the program compile to fail. You can use AVR's implicit assignment conversion with *String data types only.

ToString()

All AVR data types offer a ToString() method. Seeing ToString() in action is the best way to understand it. Consider the following code:

```
DclFld x        Type( *Integer4 )
DclFld StrValue Type( *String )

x = 77
StrValue = x.ToString()
```

which results in StrValue being assigned "77." ToString(), as you can see, is a very handy way to get a string representation of a field value. As you learn more about AVR, as has been hinted at previously, ToString() is good not only for converting types into strings, but also for formatting those strings. For example, this code:

```
DclFld x        Type( *Packed ) Len( 12, 2 )
```

```
DclFld StrValue Type( *String )

x = 123487.23
StrValue = x.ToString( “#,###.00” )
```

Results in the string value “123,487.23” being assigned to StrValue. As you can see, ToString() makes a very nice alternative to RPG’s edit codes and edit words (and their related built-in functions). Keep your antennae up for ToString(), you’ll see it used in many places.

You might wonder where ToString() came from. After all, its syntax is a bit mysterious. We don’t want to get into the complete nuts and bolts of ToString() just yet, but let it register with you that ToString() is an example of the power of object-oriented programming. OO packages data and actions into cohesive and poweful data types. In the case of ToString(), we can see that one of the actions that virtually all objects have is the ability to express themselves as strings. In green screen RPG, a packed field is just that—a packed field that represents a simple value. In AVR for .NET, though. you’ll that a packed field is really an instance of an object that in addition to offering a value, provides other related data and actions for itself. Virtually all objects do this and you’ll learn more about it as you learn more about AVR.

AVR offers alternatives to RPG’s traditional indicators

AVR offers several alternatives to both RPG’s traditional indicators and its built-in-functions (BIFs). For example, you could write this:

```
Chain Cust NotFnd( *IN50 )
If ( NOT *IN50 )...
```

or this

```
Chain Cust
If ( NOT %FOUND( Cust) )...
```

or this

```
Chain Cust
If ( Cust.IsFound )...
```

Although indicators and BIF’s are supported their use for new code is discouraged. The indicator method lends itself easily to problems because the indicator is a global variable (bringing along with it the potential grief that any global variable offers). It’s just good defensive coding to avoid indicators wherever you can—and, AVR lets you avoid them everywhere! In this example, you see that the Cust file has a property called IsFound that indicates the success of the preceding Chain operation. You’ll learn later that AVR programs are built of many objects that offer many properties. These object properties are one of the ways you can avoid using indicators in AVR.

Using RPG built-in functions are less problematic from a defensively point of view. But, as you read this document, you’ll soon discover that AVR and the .NET Framework offer power alternatives to most green-screen RPG BIFs. Having said that, remember that if you want to use BIFs (and even indicators), you certainly can.

For VB/C# programmers, RPG indicators are class-level Boolean variables used to indicate program state. There are 99 of these variables accessible in a special array called *IN (in addition to these 99 indicators, there were also a handful of other special-case indicators). Years ago, before RPG acquired “structured” operations such as IF and DOW (DoWhile), RPG programmers used global variables to manage program state. As explained in the paragraph above, except for backwards compatibility with old code, indicator usage should generally be considered a deprecated language feature.

AVR allows long variable names

Unlike traditional variable naming rules, AVR allows long variable names. While 15 or 20 characters may be the most rational maximum, AVR allows any length variable name. Long naming applies not only to variable names, but also to control names, arrays and array indexes and subroutine names.

After having lived with the 10-character (and longer ago, 6-character!) restraints green-screen RPG

imposes on field names, you'll find AVR's long variable names quite helpful.

AVR operations are always listed first

AVR operations are always listed before the operation's operands. For example,

```
Chain File( Cust ) Key( CustNo )
```

or

```
Do FromVal( 1 ) ToVal( 10 )
```

This may be initially off-putting to long-time RPG coders, but most quickly realize that having the "verb" first increases comprehending the code.

AVR programs don't have executable "mainline" code

In an ILE RPG or RPG/400 program, mainline code is that code provided between the top of the source code and the first subroutine. Any calculation specs provided here were executed once when the program was initially loaded. The essential building block of an AVR program is a class (more about that later) and classes don't have executable mainline code. That may seem quite limiting but you will soon learn that AVR has more than one trick up its sleeve to provide the semantic equivalent of mainline code. (Sneak peak: If you're even conversationally familiar with classes, you may be familiar with what a constructor is. The constructor is one way to achieve "mainline code" execution.)

Despite not having executable mainline code, AVR does continue the RPG idiom using the "mainline" area as the place to declare anything that should be globally available to all other routines in that member. Like RPG, you must declare all F-Specs, KeyLists, and constants in this area.

Local variables

Unlike traditional RPG, AVR lets you define program variables scoped only to the subroutine in which they are declared. Locally scoped variables add substantially to the robustness of your code. With locally-scoped variables, you no longer have to worry about other subroutines inadvertently changing a variable's value. With locally scoped variables your subroutines are more loosely coupled to the rest of the routines in your program (and that's a good thing!). When a routine is more loosely coupled to a program, changes made to the routine are less likely to cause problems in other parts of your code.

AVR's DclFld operation is used to declare variables. You can use DclFld in either the mainline part of the program (remember, AVR has a mainline part of code that, although it can't have any executing code, it is used to declare things global to the rest of the program.) or in subroutines. While AVR still lets you declare variables on the fly (as result fields with operations like Z-ADD) that practice is strongly discouraged; it's much better to use DclFld to declare all of your program or routine-level variables.

Briefly a DclFld looks like this:

```
DclFld x Type( *Boolean )
```

The DclFld's lineage owes partly to RPG's D-Spec and partly to CL's DclVar operation. The DclFld declares a variable's name and its data type. RPG/CL programmers should quickly see its relationship to either RPG's D-Spec and CL's DclVar. VB coders would see the above in VB as:

```
Dim x as Boolean
```

And C# coders would see it as

```
boolean x;
```

For VB/C# coders, it's important to remember that AVR does not support block-level variable scoping like VB and C# do. In AVR, variables are scoped either to the class or to a routine.

In addition to local variables, AVR also introduces several new data types. In addition to the expected

types of packed, zone, binary, and character (to name a few) AVR also introduces strings, Booleans, and integers (to name a few!). Beyond data types, there's still plenty more to cover with DclFld—as you'll soon see.

Subroutines

Subroutines are the basic build block of an RPG program. In AVR for .NET, you'll see that this is still true—AVR supports subroutines just like green-screen RPG does. For example, this subroutine,

```
BegSr MySubr
    … do some code
EndSr
```

provides a programming unit just as RPG/400 or ILE RPG programmers are familiar with. It can be called as RPG/400 or ILE RPG would have with

```
ExSr MySubr
```

but AVR also allows this streamlined syntax to call a subroutine:

```
MySubr()
```

The parenthesis denote "subroutine"—anytime an identifier is followed by parenthesis it is assumed to be a subroutine (or function, which will be covered shortly). Although using ExSr is legal and valid in AVR, this second way is the preferred way to call a subroutine in AVR. ExSr should be considered a deprecated operation code for new programs.

Why is the second method preferred? It has to with passing arguments to subroutines. AVR lets you pass arguments to subroutines—very much like you may be familiar with passing arguments to an OS/400 program using CALL/PARM. Let that concept soak in for a minute. Empowering subroutines with parameters dramatically reduces the global "knowledge" a subroutine must have about your program. Using parameters to get data into a subroutine more loosely couples those routines to your program that does their need for using global values to get data. Let's consider an example.

The following subroutine requires two parameters, x and y.

```
BegSr MySubr
    DclSrParm x Type( *Integer4 )
    DclSrParm y Type( *Boolean  )

    … do some code
EndSr
```

As you can see, the AVR DclSrParm operation is used to define subroutine parameters. The DclSrParm operation must follow the BegSr operation (or, the soon-to-be explained BegFunc operation). The DclSrParm operation will covered in more detail later; for now just understand that each DclSrParm describes a parameter's and data type, for a subroutine. You'll also learn later that subroutine parameters can be passed by value or by reference.

MySubr can be called two ways. The first of which is the verbose method which uses Exsr and the DclParm operation:

```
ExSr MySubr
DclParm a
DclParm b
```

The DclParm's must be specified in the order in which the target parameters are defined in the subroutine itself (in other words, variable a must be an integer and b must be a Boolean). Like passing parameters to an OS/400 program with Call/Parm, variable names don't have to match between the call and the subroutine definition.

Using the verbose calling method requires a single line for the Exsr and additional single lines for each DclSrParm. A more succinct way, and the recommended way, of making the call is:

```
MySubr( x, y )
```

There isn't anything wrong with the verbose method. But as you can see, this second method is cleaner and more concise. In either case, a and b are passed as arguments to MySubr. If the subroutine doesn't have parameters, you simply use:

```
MySubr()
```

Take care to provide the trailing, closing parentheses. These parentheses are very important. These parentheses are what indicates to the AVR compiler that this line is calling a routine. Without the parentheses the line wouldn't compile.

C# coders should recognize an AVR subroutine as the semantic equivalent of a C# void function and VB programs should recognize it as a VB procedure.

The previous section said that subroutines are the basic building block of an RPG program. That may have been true in the green-screen RPG world, but in AVR, there is another basic build block called a function. As you'll see in this section, functions are just as important as program building blocks as subroutines are.

Constants

AVR constants let you declare program values that cannot change during program execution. Constants aid program readability and add compile-time protection against silly typing mistakes. For example, consider the need to create a viewstate variable for an ASP.NET application (an ASP.NET viewstate variable is a way to persist data from one ASP.NET page to another). Here's how you'd save a field named CMName to a viewstate variable that you named "customername."

```
Viewstate[ "customername" ] = CMName
```

To fetch the value, you'd use this code (If you aren't familar with ToString(), don't worry about if for now):

```
CMName = Viewstate[ "customername" ].ToString()
```

The line above works just fine. However this line fails:

```
CMName = Viewstate[ "CustomerName" ].ToString()
```

because viewstate keys are case-sensitive. Recall that I earlier said that AVR isn't case sensitive; however there several .NET data structures for which the key values are case-sensitive. Think of these key values as case-sensitive, alphameric array indices. Adding to the frustration that these keys are case sensitive is that errors like the one above, where "CustomerName" was provided when "customername" should have been provided, can't be caught by the compiler. It's up to diligent runtime testing to prove the existance of the error.

Constants can help you out of this case insensitive quagmire. Consider this code:

```
DclConst CUSTOMERNAME Value( "customername" )

Viewstate[ CUSTOMERNAME ] = CMName
```

Using the constant CUSTOMERNAME (which gets substituted at compile time for the value "customername") protects you from case errors. Because AVR itself isn't case senstive, you could have also coded the assignment above as:

```
Viewstate[ customername ] = CMName
```

and the code will compile and work as expected. Note that even though AVR isn't case senstive, it's a good convention to always code your constants with upper case names so you that you easily distringuish them from variables in your program. Also, if you'd happen to misspell the constant, the conpiler would catch that error for at compile time.

Functions

Functions are similar to subroutines, but a function always returns a single value. If you're familiar with

RPG BIFs (built-in functions), AVR functions provide a similar capability but you're in control what they do. Let's first look at a built-in function example as a refresher to what a function does. We'll use the %SUBST BIF as our example. Consider the code below:

```
If ( %SUBST( CustName, 1, 1 ) = "A" )
    … Customer name started with "A"
Else
    … some other code
EndIf
```

This code uses the RPG %SUBST BIF to determine if the first character of a variable named CustName is an "A." The function returns a string value. Assuming the customer's name did start with an "A," the %SUBST call would have resolved to an "A," resulting in the test ultimately looking like this to the compiler:

```
If ( "A" = "A" )
```

The concept of "resolves to" is very important here. The call to %SUBST resolved to "A" at runtime. This is an important feature of functions—they are resolved inline and their results can be used in expressions. For example, let's say you wanted to know if a customer's name begins with "AL." You could use this code to do that:

```
If ( %SUBST( CustName, 1, 1 ) + "L" = "AL" )
    … Customer name started with "AL"
Else
    … some other code
EndIf
```

This illustrates that a function's return value (what it resolves to) can be used in an expression. In other words, the results of a function can be used inline without intermediate variables. In the example above, the test first resolves to this code at runtime

```
If ( "A" + "L" = "A" )
```

Then, immediately to this:

```
If ( "AL" = "AL" )
```

In AVR, you'll create functions to return values to your program using your data and the context of your application to determine what should be returned (and why!). Let's take a look at a simple example. The AVR function below calculates a 5% sales tax on a sale amount. The sales tax calculated is the value this function returns.

```
BegFunc CalcSalesTax Type( *Packed ) Len( 12, 2 )
    DclSrParm SalesAmount Type( *Packed ) Len( 12, 2 )

    DclFld TaxAmount Type( *Packed ) Len( 12, 2 )

    TaxAmount = %DECH( SalesAmount * .05, 12, 2 )

    LeaveSr TaxAmount
EndFunc
```

Let's take a look at each line of code. The first line declares the function. This does two things: it names the function (CalcSalesTax, in this case) and identifies its return type. All functions must have a return type specified in their declaration.

```
BegFunc CalcSalesTax Type( *Packed ) Len( 12, 3 )
```

The next line declares a parameter for the function. In this case, the sales amount is being passed to the function. Functions can have any number of parameters. Note that the parameter types don't have anything to do with the return type. In this case they are the same, but they don't have to be (and often aren't).

```
    DclSrParm SalesAmount Type( *Packed ) Len( 12, 3 )
```

The next line declares a return value as in intermediate variable. This line isn't necessary but is added for

clarity. A subsequent example will show how this line can be eliminated.

```
DclFld TaxAmount Type( *Packed ) Len( 12, 3 )
```

The next line performs the calculation. In this case the function uses the %DECH BIF to calculate the rounded sales tax amount. As with most of AVR's BIFs, the syntax is identical to that of the corresponding ILE RPG function.

```
TaxAmount = %DECH( SalesAmount * .05, 12, 2 )
```

The next line returns the value of the function. Every function must have at least one LeaveSr.3 LeaveSr's job is to cause program flow to immediately leave the function, returning the result value specified. VB and C# programmers will recognize LeaveSr as performing the same function as VB and C#'s Return statement.

```
LeaveSr TaxAmount
```

And finally, the function is ended with EndFunc.

```
EndFunc
```

Here's how the preceding function could have been written without the TaxAmount:

```
BegFunc CalcSalesTax Type( *Packed ) Len( 12, 2 )
    DclSrParm SalesAmount Type( *Packed ) Len( 12, 2 )

    LeaveSr %DECH( SalesAmount * .05, 12, 2 )
EndFunc
```

In the example above the sales tax is calculated and returned all at once. This style of function saves code and may be appropriate for very simple functions. However for most functions I think you'll find that using an intermediate variable for the return value adds substantially to the clarity of the function.

There is a subtle nuance that the CalcSalesTax function teaches: It receives an input and returns an output. This function knows nothing about the rest of the program, nor does it need to—for all intents and purposes CalcSalesTax is a black box that hides its implementation from the rest of the program. All a consumer need to know about the function is that when you pass it a value, it returns a 5% sales tax. The consumer doesn't know, or care, how the sales tax was calculated.

Long-time RPG coders should think pretty deeply about the concept of CalcSalesTax not having any dependency on global variables. This is a good thing! With CalcSalesTax's action isolated from the rest of the program we know it's 100% safe to make changes to CalcSalesTax without causing any unexpected side effects throughout the rest of the program. This black box concept, known also as information hiding or loose coupling is key to quality programs. You'll see this concept revisited many times as you learn about AVR.

Summary

This document introduced you to a number of AVR's unique features. For veteran RPG programmers, concepts were covered that you've never seen in RPG before. There are certainly a lot of new and exciting features in AVR. And, as you'll see as you learn more about AVR, that AVR has taught the old RPG dog many other new dazzling tricks (OO, anyone?). However, you'll also see that AVR, for all its newfound power and capabilities, is surprisingly backwards compatible with many time-honored RPG features and facilities. For example, virtually all of green-screen's RPG file IO model persists. You'll also recognize many other old RPG friends as you read on. For VB/C# coders, this document introduced AVR's syntax and also hinted at the fact that AVR does indeed, semantically if not syntactically, have a lot in common with VB.NET and C#.

As you learn AVR, you'll quickly see that RPG isn't near as old fashioned, or foreign, as you may have once thought.

This page is intentionally left blank.

Programming with classes

Classes are the building blocks of AVR for .NET programs. Nothing happens without them! Classes provide a way to package your subroutines and data into cohesive units; classes provide a way to extend your applications in non-disruptively; classes let you isolate processing; classes let you enhance and extend existing code; in short, classes let you do lots of things. While it's possible to write a simple AVR for .NET without knowing much about classes, however it is impossible to write an effective, enterprise application without a solid knowledge of classes. This text introduces you to using classes with AVR for .NET

What are classes?

Classes are groups of subroutines (known formally as methods) and data (provided by AVR elements such as DclDiskFile or DclFlds) organized to perform a well-focused group of tasks (ie, add or change a customer, process a purchase order, or validate data against business rules). For example, when you create a Windows or Web application, you'll notice that the code-behind for a Windows or Web form provides a class (denoted by BegClass and EndClass) for the form operations. The code-behind class for a Windows or Web form is a very typical class; it provides data and actions related directly to the operation of the form. OO calls this concept "encapsulation"; the code-behind class encapsulates, or compartmentalizes, form-related actions and data. When other tasks are necessary, such as performing IO or business rules, ideally those tasks would be shuffled off to their classes (which, again, tightly compartmentalizes a given set of operations and data).

The concept of using encapsulation to isolate related chunks of code into their own little worlds is almost the polar opposite of the inherent everything-is-global programming mindset of many a veteran RPG programmer. The value of partitioning your programs into classes won't be apparent immediately, but you'll learn (more quickly than you think!) that the divide-and-conquer method allowed by classes is perhaps your best weapon against the intrinsic complexity of modern enterprise applications. Once a class is finished and tested, it's available to use time and time again.

Essentially another way to say "information hiding," encapsulation is the act of hiding as much of a class's implementation as possible. The consumer doesn't know, doesn't need to know, and doesn't care, how something happens inside a class; it just needs to know what happens (ie, adding a customer, updating payment history, etc). The more any one class can hide from the others, the better. The less each class knows about each other, the lower coupling between classes (and the lower the coupling the better).

Contrast low coupling between classes with the high coupling of RPG /400 subroutines. RPG/400 subroutines don't provide an interface with which to talk to each other. Communication between these subroutines must be done with global variables. As a result, you're never absolutely sure of what the ripple-down effects are of changing the code in any one subroutine. Effective encapsulation of class data helps solve the curse of global data.

In some ways, you can think of a class as having most of the behavior of an old-time RPG program. Both a class and an old RPG program have a declaration area (the area before any subroutines are declared that provides global declarations for the rest of the class or member and both have subroutines. A difference

between a class and an old-school RPG program is that a class doesn't allow any executable mainline code before the first subroutine is declared (you'll later learn, though, that a class's constructor provides an end-run to this limitation). Usually a single source member comprises a class—very rarely would a single souce member define more than one class.

Here is a tiny example of a class. This class provides a silly start at encapsulating a simple calculator. After calling the Multiply method(), the results of that operation are available in the Answer field. This is hardly a well-designed or complete class, but it does shown show you some of the elements of a class (it encapsulates data and an action for a focused task).

```
BegClass Calculator
    DclFld Answer Type( *Integer4 ) Access( *Public )

    BegSr Multiply Access( *Public )
        DclSrParm Value1 Type( *Integer4 )
        DclSrParm Value2 Type( *Integer4 )

        Answer = Value1 * Value2
    EndSr
EndClass
```

From an object oriented perspective, a way that classes can be defined is that they are "blueprints" for objects. Think chocolate chip cookie recipe and chocolate chip cookie. The recipe is the class and the actual cookie is the object. The object is the tangible, usable part of a class. In OO parlance, the cookie is an instance of the recipe. An object is an instance of a class.

AVR for .NET uses classes in two contexts: implicitly and explicitly. Implicit classes are those that are automatically created for you (by Visual Studio) and directly associated with a Windows or Web form. Implicit classes are know as code-behind classes. Explicit classes are secondary classes that you create. Explicit classes aren't directly associated with a user interface. Secondary classes are generally packaged into their own binary (a DLL) so that they can be reused across many projects.

The rest of this document explains classes and their construction and use in more detail.

Instancing classes

A class must be instanced, or newed, before it can be used. Shared members (more on them later) do provide an exception to the rule for instancing classes—but shared members are a special case and most of the time you will need to instance your classes explicitly. When a class is instanced, its owning class is known as its *consumer*. Any reference in this document to a class's consumer means that class's parent class.

Here are a few considerations for instancing a class.

Declare a class instance:

```
DclFld mc1 Type( MyClass )
```

Declare, and new, a class instance:

```
DclFld mc1 Type( MyClass ) New()
```

Declare, and new, a class instance passing parms to its constructor:

```
DclFld mc1 Type( MyClass ) New( 128 )
```

Declare a class with deferred new:

```
DclFld mc1 Type( MyClass )
.
.
.
mc1 = *New MyClass()
```

Declare a class with a deferred new and pass its constructor a parameter:

```
DclFld mc1 Type( MyClass )
.
.
.
mc1 = *New MyClass( 128 )
```

Class scope

A class instance has a scope—just like any other variable. That scope can be global to a class (ie, declared in the class before any methods—what an RPG programmer may know as the *mainline* part of the class) or local to a subroutine or function.

Class declared and instanced globally

```
BegClass MyMainClass
    DclFld mc1 Type( MyClass ) New()

    BegSr MyRoutine1
       // mc1 is instanced and
       // available to this method.
    EndSr

    BegSr MyRoutine2
       // That same instance of mc1 is
       // also available to this method.
    EndSr
EndClass
```

Because the class instance mc1 is declared in the class's mainline, it is available to all the subroutines (or functions) in the class.

Class declared globally but instanced on demand

```
BegClass MyMainClass
    DclFld mc1 Type( MyClass )
    .
    .
    .
    BegSr MyRoutine
        If ( mc1 = *Nothing )
            mc1 = *New MyClass()
        EndIf
    EndSr
EndClass
```

Why might you want to defer the instancing of a class? Classing instancing is a relatively costly process—not to mention the memory required by the class instance.

There isn't any point in instancing a class if it's not needed. There may be a path through the application that doesn't need a given class. Thus, for conditional classes, it saves resoruces to defer their instancing until needed.

Most, though not all, objects declared globally will be instanced upon their declaration and that instance persists for the life of the parent class. Objects declared locally persist only for the lifetime of the routine in which they are declared.

Remember, too, that any class for which you want to use its events—by declaring the class with WithEvents(*Yes)—must be declared globally

Using constructors

Constructors are special-case subroutines that are implicitly performed when a class is instanced. This code is performed only when the class is instanced—it's not called otherwise. For RPG programmers, a constructor behaves a little like RPG's *INZSR routine; it also performs a little bit like RPG mainline—its code is only performed once, implicitly. Typically you provide variable initialization code and other start-up type code in a constructor. Constructors can optionally use parameters, and, as you'll soon see, a class can have more than one constructor. When using multiple constructors each constructor must be distinguished by a unique parameter list. More on the topic of "overloaded" constructors in a bit.

Here is a small class with a basic constructor. This constructor doesn't have any parameters.

```
BegClass MyClass
    .
    .

    BegConstructor Access( *Public )
        // Some code.
    EndConstructor

EndClass
```

Consumer code to use MyClass could look like this:

```
DclFld mc1 Type( MyClass ) New()
```

When MyClass is instanced (in this case on the declaration line), the code inside the constructor is performed. If the class instancing were deferred, as below, the code in the constructor is performed when the class is instanced.

```
DclFld mc1 Type( MyClass )
.
.
.
mc1 = *New MyClass()
```

A constructors without parameters is known as the default constructor. If you don't explicitly provide a default constructor, an empty default constructor is assumed by the compiler.

Constructors with parameters

Here is a class with a constructor that has one parameter.

```
BegClass MyClass
    .
    .

    BegConstructor Access( *Public )
        DclSrParm Counter Type( *Integer4 )
        // Some code.
    EndConstructor
EndClass
```

When this class is instanced, a single integer must be passed to the constructor. For example,

```
DclFld mc1 Type( MyClass ) New( 77 )
```

or

```
DclFld mc1 Type( MyClass )
.
.
mc1 = *New MyClass( 77 )
```

In both cases, a single integer value is being passed to the constructor. If the constructor took more than one parameter, these parameters would be specified as a comma-separated list. For example,

```
DclFld Mc1 Type( MyClass ) New( p1, p2, p3 )
```

or

```
DclFld Mc1 Type( MyClass )
.
.
Mc1 = *New MyClass( p1, p2, p3 )
```

Calling constructors is just like calling subroutines and functions--the number and types of parameters must match and must be passed.

Using constructor parameters

In addition to performing an one-time, startup processing, constructors are often also used to set class-level variable values.

```
BegClass MyClass
    .
    DclFld Counter Type( *Integer4 )
    .

    BegConstructor Access( *Public )
        DclSrParm Counter Type( *Integer4 )

        *This.Counter = Counter
    EndConstructor
EndClass
```

In the code above, MyClass has a class-level variable named Counter. This variable is available to all other methods of MyClass. Notice that the class-level Counter variable above is private. As the class is written above, the only way that variable can be changed by a class consumer is through the constructor. When the constructor is called, it is a passed a single integer value (named Counter) and that parameter value is used to set the class-level variable Counter. Note that within the constructor (or any method for that matter) when a method-level variable has the same name as a class-level variable, the class-level variable must be qualified with *This to indicate which of the two like-named variables you're referencing.

Do not let this construct confuse you! In this case, the two Counter variables don't have anything to do with each other! Some RPG programmers would rather code the above routine as shown below. Note the use of the at-sign (@) to differentiate the two Counter variables.

```
BegClass MyClass
    .
    DclFld Counter
    .

    BegConstructor Access( *Public )
        DclSrParm @Counter Type( *Integer4 )

        Counter = @Counter
    EndConstructor
EndClass
```

Contriving unique names with the @ sign or another special prefix or technique is not the preferred syntax. You don't need to contrive unique names; qualifying the class-level member is a much better technique and solves the problem gracefully. Read the code like this like this, "the constructor-level variable named Counter (owned by the constructor), is setting the class-level variable (owned-by the class) named Counter."

Overloaded constructors

When a class offers more than one constructor, that's known as *overloading* the constructor. Overloaded constructors offer optional ways to instance a class. The class below has two constructors, one that accepts no parameters, one that accepts one parameter.

```
BegClass MyClass

    BegConstructor Access( *Public )
         .
    EndConstructor

    BegConstructor Access( *Public )
        DclSrParm Counter Type( *Integer4 )
          .
    EndConstructor

EndClass
```

RPG programmers new to .NET often get a little unnerved by the concept of overloaded constructors. For what it's worth, RPG coders are familar with the concept of overloading and don't even know it! Can you think of an RPG example of overloading? Hint: How many arguments does the %SUBSTR built-in function need?

While on the subject of overloaded constructors, ponder this amazing .NET fact: you can also overload AVR's subroutines! That means you could provide two different subroutines, with the same name, differentiated by their parameter list.

The class above can be instanced with either

```
DclFld Mc1 Type( MyClass ) New()
```

or

```
DclFld Mc1 Type( MyClass ) New( 77 )
```

The value of an overloaded constructor is illustrated nicely with the .NET Framework's System.Text. StringBuilder class. It can be instanced like this:

```
DclFld sb Type( System.Text.StringBuilder ) New()
```

or like this:

```
DclFld sb Type( System.Text.StringBuilder ) New( 128 )
```

The first example creates an instance of the stringbuilder with the default buffer size of 16 bytes. The second creates a stringbuilder with a buffer of 128 bytes. Overloaded constructors add flexibility in how classes are instanced.

Overloaded constructors—chaining constructor calls

Here's an advanced technique, but one that has merit in some situations. Assume you've got a class with a default constructor and one constructor that has one parameter.

```
BegClass MyClass
    DclFld Counter Type( *Integer4 )

    BegConstructor Access( *Public ) This( 24 )
         .
    EndConstructor

    BegConstructor Access( *Public )
        DclSrParm Counter Type( *Integer4 )

        *This.Counter = Counter
    EndConstructor
EndClass
```

To instance this class with the constructor that takes one integer parameter, you'd use

```
mc = *New MyClass( 36 )
```

to set Counter to 36. To instance the class with the default constructor, like this:

```
mc = *New MyClass()
```

Instances the class with the default constructor, but the This(24) tells the default constructor to call the other constructor passing it a value of 24. When constructors are chained like this, the constructor being chained is called first, then the original constructor's code is performed.

Beware no explicit default constructor

When you provide a class with no default constructor, one is implicitly created for you. For example, this class

```
BegClass MyClass
    .
    .
EndClass
```

can be instanced with

```
DclFld mc1 Type( MyClass) New()
```

where the New() calls the implicitly created default constructor. However, if later you go back and add a constructor that takes parameter(s) , like this:

```
BegClass MyClass

    BegConstructor Access( *Public )
        DclSrParm Counter Type( *Integer4 )
          .
    EndConstructor
EndClass
```

Then no default constructor is implicitly created for you and the class must be instanced with the explicit constructor you provided. This means that any code you have using this class that instances it like this:

```
DclFld mc1 Type( MyClass) New()
```

That code is now broken (because the only constructor the class has now requires one integer parameter). For this reason, you should also implicitly provide a default constructor, even if its code is empty.

Class members

Classes talk to each other through the class's interface. Subroutines, functions, fields, properties, and events comprise a class's interface. Don't *confuse* the use of interface here with "user interface." The kind of interface we're talking about here is a programming interface. That is, the parts of a class that are marked public (or protected, more on that later) are shared with the outside world (the class's consumers) and define its programming interface.

The four member types that a class can include are:

- **Methods** - provide actions to work against class data (subroutines and functions). The constructor is a special-case method.
- **Fields** - provide direct access to class data.
- **Properties** - provide protectable, filterable, read-only, write-only, or read/write access to class data.
- **Events** - provide "exit points" where the consumer can add code to be remotely executed.

Each member has a specific visibility (which controls how exposed, or seen by consuming classes, any one member is). This visibility is controlled by the member's *Access* keyword The most frequentely used values are **Public* (seen by the consumer) and **Private* (not seen by the consumer). Each member has an owner. The owner of most members is the member class's current instance. Members owned by the current object instanced are marked *Shared(*No)* (which is the default).Nonshared members aren't available until the class is instanced. Members owned by the class itself, rather than an instance of the

class, are marked *Shared(*Yes)*. Shared members are available without instancing the class. In nearly every case, your class members will be *Shared(*No)*. We''ll cover shared members more later.

Let's take a closer look at each member type a little more closely.

Methods

Methods are subroutines or functions offered by the class to perform actions. Except for specifying visibility and owner, subroutines and functions work just as they in procedural RPG.

Fields

Fields provide a consumer unencumbered access to values in a class. This class has two fields, one public and one private. Private fields are most often used to provide working storage inside the class. Public fields are a cheap and quick way to provide access to a data member—all that's required is to declare the field public and it's seen by any consumer classes.

```
BegClass MyClass
    DclFld SaleDate   Type( *Date ) Access( *Public )
    DclFld x          Type( *Packed ) Len( 12, 2 ) Access( *Private )
    .
EndClass
```

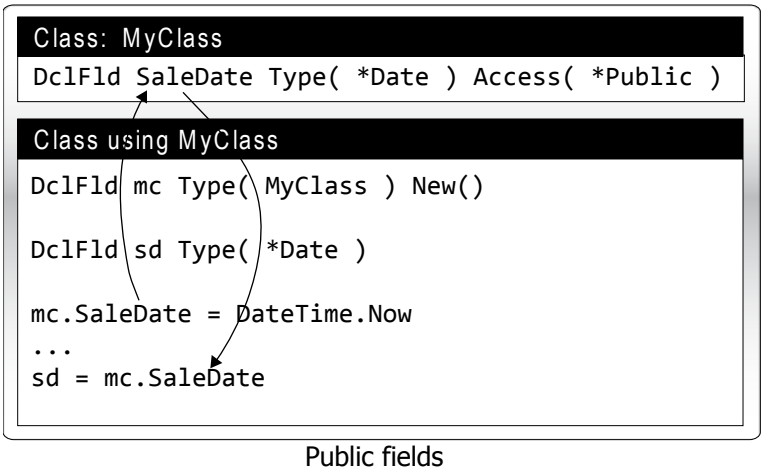
Using the public field, in this case *SaleDate*, might look like this:

```
DclFld mc         Type( MyClass ) New()
DclFld sd         Type( *Date )

// Set the public field.
mc.SaleDate = DateTime.Now

// Get the public field.
sd = mc.SaleDate
```

The private x variable belonging to MyClass is not available to MyClass's consuming class.. The consumer knows absolutely nothing of MyClass's x. member because it is private (which, by the way, is the default access).

Use public fields sparingly

The problem with public fields is that they provide unencumbered read/write access to class data. That is, the class owning the public field doesn't know when a consumer changes the field's value. Further, the parent class can't impose any restrictions on the value assigned to the public field. For example, let's assume that the SaleDate value in the class cannot have weekend date values (ie, no Saturday or Sunday dates). Because the



Public fields

Public fields provide unconditional read/write access to the field value. This direct read/write access could be a problem because it provides unchecked access to the public field—offering the class no way to know when the value is changed.

SalesDate is a field the consumer can put any value into the field it wants and there isn't any In this picture, the *mc* instance of MyClass provides unemcumbered read/write access to its SaleDate public field. The class doesn't know when the SaleDate value is changed nor does it know when its value is read.

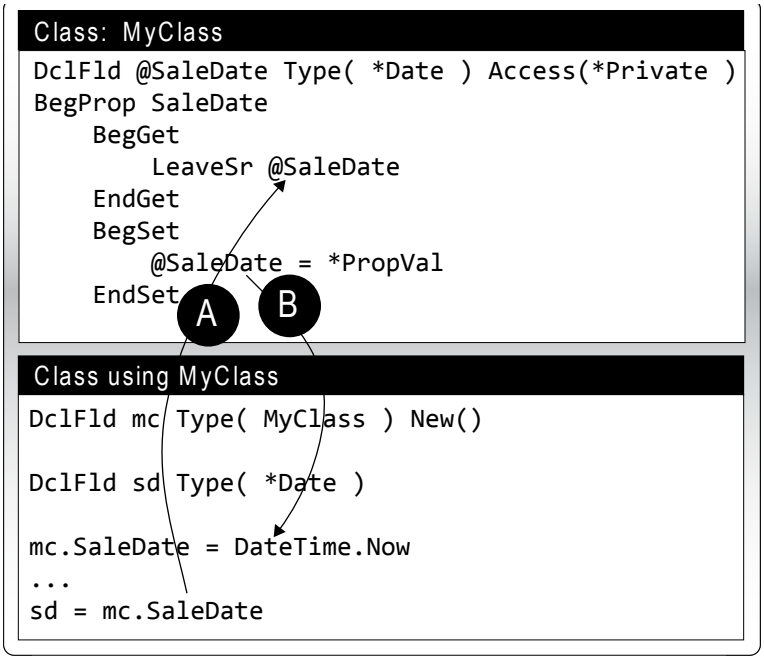
A further problem with public fields is that they can only be read/write—they cannot be read-only. Thus, the class isn't able to guarantee that the public field value can only be read from a consumer. Consumers can read, and change, any public field.

That's bad!

Public fields are the quick and dirty way for a class to provide member data. Using properties is a much better option. Read on.

Using properties

Properties are the better alternative to fields for classes to provide class data to the consumers. However, like many better things, they take more work than their lesser equivalents. Unlike a public field, properties disconnect the value of the property from the get/set access points. To get a property value, to read it, requires a BegGet/EndGet routine; to set it, to write to it, requires a BegSet/End routine. These two routines are enclosed inside a BegProp/EndProp construct. Note that either the Get or Set routines are optional. That is, to omit the Get routine makes the property write-only and to omit the Set routine makes the property read-only.



A property in action

Compare the way SaleDate is accessed in this figure to the way public fields worked on the previous page. From the consumer's point of view, there is no difference. But internally, to the class, there is a big difference.

The picture above shows a typical property in action. Typically, a property needs to have an associated storage variable for it to, well, store, its value. In this example, the variable @SaleDate is that storage variable. Note that this variable is private—its value is *not* directly available from a consumer of this class.

The property name is provided by the BegProp statement. Usually you'll want some convention that associates this property with its storage variable. In this case, the storage variable is the same name as the property—prefixed with an @ sign.

Getting the @SaleDate value (action A)

The "get" of the @SaleDate variable is done with the code inside the BegGet/EndGet. If you're familar with functions, this code should mostly make sense. The following line of code in the consumer

```
sd = mc.SaleDate
```

goes through the BegGet/EndGet to fetch the value of @SaleDate from the class instance.

Setting the value of @SaleDate (action B)

The "set" of the @SaleDate variable is done with the inside thie BegSet/EndSet. Consider this code in the consumer to set the SaleDate property:

```
mc.SaleDate = DateTime.Now
```

This code implicitly calls the BegSet routine, passing the value on the right side of the = sign. This passed value is seen to BegSet with the keyword *PropVal

So what! What do properties buy you?

Given the example we just looked at, it's easy to see why you might think that properties aren't worth the trouble over using public fields. In the example we just looked at, the public property SaleDate is semantically identical to the public field SaleDate. And, in fact, given the way the SaleDate property was written it doesn't buy you very much over using a public field. Let's take a look at how the property code can be tweaked to make the property more powerful than the field.

Making SaleDate a read-only property

The first way that properties can improve things is by offering the ability to make read-only properties. This is easily achieved by simply omitting the BegSet/EndSet from the property definition, as shown below.

```
DclFld @SaleDate Type( *Date )

BegProp SaleDate Type( *Date )  Access( *Public )
    BegGet
        LeaveSr Value( @SaleDate )
    EndGet
EndProp
```

In this case, you might wonder how SaleDate property ever gets set. It doesn't, but the @SaleDate field can be—and setting that inside the class sets what the SaleDate property returns. Understanding this concept is key. In this case, MyClass sets the @SaleDate field to determine the value that the SaleDate property returns. It's none of the outside world's business how the *SaleDate* property it got set. Although less frequently done, you could also provide a write-only property by including the BegSet but excluding the BegGet.

Also note that while you'll frequently use a dedicated storage variable, as is the case with @SaleDate in this example, it's not uncommon for a property's Get method to return something intrinsically available in the class. For example, you could surface a file's EOF status with:

```
BegGet
    LeaveSr %EOF( Cust )
EndGet
```

Performing other code during the get and/or set

Because BegGet is just a special case function (it leaves with a value) and BegSet is a special case subroutine (it does not leave with a value), you can add any code you need to fire before the storage value is fetched or before and/or after it is set.

```
DclFld _SaleDate Type( *Date )
BegProp SaleDate Type( *Date )  Access( *Public )
    BegGet
       // You can perform any code here before
       // the @SaleDate variable is fetched.
      LeaveSr Value( _SaleDate )
    EndGet

    BegSet
        // You can perform code here before the
        // @SaleDate variable is set.
         _SaleDate = *PropVal
```

```
        // You can performan code here after the
        // @SaleDate variable is set.
    EndSet
EndProp
```

As you can see, making data available in your classes as properties instead of fields requires more work. However, the extra work provides more control and capabilities.

Events

Events are a paradox. They are a) a hard concept to get at first but b) once you understand them, they are actually pretty simple and make a lot of sense. The class below provides an event called AfterSum. This event is raised immediately after the Sum function's Result is calculated.

The DclEvent AfterSum is what actually declares the event. Note the call to the AfterSum() method near the end of the Sum function. The DclEvent AfterSum tells the compile to not bark about the absence of an actual AfterSum method. If needed, a consumer will provide the actual method to be called when the AfterSum() call occurs.

```
Using System

BegClass Events Access(*Public)

    DclEvent AfterSum Access( *Public )

    BegFunc Sum Type( *Integer4 ) Access( *Public )
        DclSrParm Value1 Type( *Integer4 )
        DclSrParm Value2 Type( *Integer4 )

        DclFld Result Type( *Integer4 )

        Result = Value1 + Value2

        AfterSum()
        LeaveSr Result
    EndFunc
EndClass
```

This class has an AfterSum event. This event is raised by its call to AfterSum(). The DclEvent says to the class, "Don't expect to find AfterSum() in this class, but there is a possibility that the consumer may provide this routine for you."

Events are optional and do not have to be handled by consumers.

The consumer code to use the class above and fire its AfterSum method is shown below. Note that the Event keyword associates the AfterSum event with the subroutine that is actually providing the code that gets called when the event is raised.

```
DclFld ev Type( Events ) New() WithEvents( *Yes )

BegSr TestSum
    DclFld x Type( *Integer4 )

    x = ev.Sum( 2, 5 )
EndSr

BegSr AfterSum Event( ev.AfterSum )
    MsgBox "Sum method just completed"
EndSr
```

This subroutine gets called when the AfterSum event is raised.

Events must be implemented as subroutines, they cannot be functions. Note that in the example above, no parameters were passed to AfterSum. Passing parameters is our next challenge.

An event with a simple parameter

This example shows an event that gets passed a parameter. The parameter (or parameters, if necessary) are declared immediately after the DclEvent. The call to event must include the parameter. In this case, the newly calculated result is being passed to the consumer.

```
Using System
```

```
BegClass Events2  Access(*Public)

    DclEvent AfterSum Access( *Public )
        DclSrParm Sum Type( *Integer4 )

    BegFunc Sum Type( *Integer4 ) Access( *Public )
        DclSrParm Value1 Type( *Integer4 )
        DclSrParm Value2 Type( *Integer4 )

        DclFld Result Type( *Integer4 )

        Result = Value1 + Value2

        AfterSum( Result )
        LeaveSr Result
    EndFunc
EndClass
```

The newly calculated result is passed to the consumer.

The client code is similar to before; the difference is that the AfterSum method must now provide the parameter it expects.

```
DclFld ev Type( Events ) New() WithEvents( *Yes )

BegSr TestSum
    DclFld x Type( *Integer4 )

    x = ev.Sum( 2, 5 )
EndSr

BegSr AfterSum Event( ev.AfterSum )
    DclSrParm Result Type( *Integer4 )

    MsgBox "The return value will be " + Result.ToString()
EndSr
```

An event with parameters: the best way

Let's assume that you need to return several values through the event handler. While you could stack up the DclSrParms, the better way is to extend the System.EventArgs class and use the extended class to pass the parameters. This allows you to follow .NET convention by providing both a *Sender* and *e* argument for your event.

```
Using System

BegClass Events2  Access(*Public)

    DclEvent AfterSum Access( *Public )
        DclSrParm Sender Type( *Object )
        DclSrParm e      Type( AfterSumEventArgs )

    BegFunc Sum Type( *Integer4 ) Access( *Public )
        DclSrParm Value1 Type( *Integer4 )
        DclSrParm Value2 Type( *Integer4 )

        DclFld Result Type( *Integer4 )

        Result = Value1 + Value2

        AfterSum( *This, *New AfterSumEventArgs( Value1, Value2, Result ) )
        LeaveSr Result
    EndFunc
EndClass
```

The parameter e is of type AfterSumEventArgs. An instance of that class is passed as one of the two event arguments.

```
BegClass AfterSumEventArgs Extends( System.EventArgs ) Access( *Public )
    DclFld Value1 Type( *Integer4 ) Access( *Public )
    DclFld Value2 Type( *Integer4 ) Access( *Public )
    DclFld Sum    Type( *Integer4 ) Access( *Public )

    BegConstructor Access(*Public)
        DclSrParm Value1 Type( *Integer4 )
        DclSrParm Value2 Type( *Integer4 )
        DclSrParm Sum    Type( *Integer4 )

        *This.Value1 = Value1
        *This.Value2 = Value2
        *This.Sum    = Sum
    EndConstructor
EndClass
```

SystemEventArgs is extended with additional members as needed. In this case, three membrs (Value1, Value2, and Sum) are added. Note the constructor requires these three values be passed upon this object's instancing.

Note how the parameter list for AfterSum now follows .NET sender/e convention. All of the values passed to subroutine are now available as members of the e variable..

The client code is again similar:

```
DclFld ev Type( Events ) New() WithEvents( *Yes )

BegSr TestSum
    DclFld x Type( *Integer4 )

    x = ev.Sum( 2, 5 )
EndSr

BegSr AfterSum Event( ev.AfterSum )
    DclSrParm Sender Type( *Object )
    DclSrParm e      Type( AfterSumEventArgs )

    MsgBox e.Value1
    MsgBox e.Value2
    MsgBox e.Sum
EndSr
```

While this method takes more code, it is consistent with the recommended .NET convention of sending an *Object and either System.EventArgs or a derivative to the event.

Shared members

Generally, the members of a class are owned by an instance of that class. Members owned by an instance of a class are known as instance members. As the name implies, a class must be instanced for its instance members to be available to a consumer. For example, consider the following class snippet:

```
BegClass MyClass Access( *Public )
    DclFld x Type( *Integer4 ) Access( *Public )
EndClass
```

To reference the x member, you must first instance MyClass

```
DclFld mc Type( MyClass ) New()
MsgBox mc.x   // Reference x instance member
```

If you tried to reference the x field directly like this

```
MsgBox MyClass.x
```

you'd get an object-not-instanced error, because x is an instance member.

In addition to instance members, classes can also have shared members. Shared members are class members owned by the class itself, not an instance of it. For example, consider this class snipppet:

```
BegClass MyClass Access( *Public )
    DclFld x Type( *Integer4 ) Access( *Public ) Shared( *Yes )
EndClass
```

In this case, x is shared (as denoted by the Shared(*Yes) keyword). You don't have to instance MyClass to use shared member x. For example, you could now do this:

```
MsgBox MyClass.x
```

and x's value would be displayed.

Shared members are available globally to a project that has access to the shared member's class. There is only ever one copy of shared member available, so you could think of shared members as super-global variables. That is, if code in one place changes a shared member's value, that change is visible to all other code in the project

What's the point?

Generally shared members are used to make methods available without needing to instance the class. For example, consider a math class that needs to offer a routine to calculate a square root.

```
BegClass MyMath Access( *Public )
    BegFunc CalcSqrRoot  Type( *Integer4 ) Access( *Public ) Shared( *Yes )
       DclSrParm Value Type( *Integer4 )
       LeaveSr Value * Value
    EndFunc
EndClass
```

In this case, CalcSqlRoot needs no other members to do its work and it's a perfect candidate for being a shared member. It could be used like this:

```
x = MyMath.CalcSqrRoot( 4 )
```

Shared member gotchas

At first blush, once you realize what shared variables can do, your initial reaction might be to say, "Wow. Shared members can do all that! I'm gonna forget all that nonsense about instancing classes and just always use all shared members." However, that's the wrong approach! Shared variables, because of their globalness, break all rules of encapsulation and should be used *very sparingly*. Usually the best candidates for shared members are methods that offer library routines as shown with the CalcSqrRoot routine. Rarely would you use shared members for fields or properties.

Shared members rules of thumb

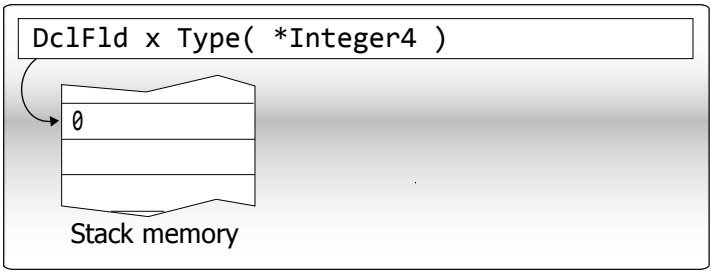
- Don't use shared members for DclDB, DclDiskFile, or DclMemoryFile objects. Because shared members are global, the behavior of these objects when shared is quite hard to predict.
- Remember that all shared members are shared by all users of Web applications. Ponder that statement a minute. It means that if you have a shared memory file in a Web app, that memory is being shared by all users of the Web app! Probably not what you intended! Except for packaging library routines, be especially wary of shared members in browser-based applications.
- In any one classe, shared members can only reference other shared members. Shared members cannot reference instance members from the same class.
- Shared members can't use the *This keyword—because *This refers to the current instance of the class and shared members don't belong to, and can't recognize, a current class instance.
- If you can't state clearly why a member should be shared, you're probably sharing it for the wrong resaons. (usually to break some sort of scope issue). Making a member shared may indeed be a quick bandaid to make a member visible to other classes, but more times than not, that quick-fix mentality will come back to bite you!

Value types and reference types

The .NET Framework provides two types of variables: value types and reference types. Understanding the differences in the two is very important.

Value Types

Value types are those variables whose base value is a number (integers, packed, dates, time, Boolean, etc). Value types are also distinguished by their fixed size (that is, a long integer is always four bytes, etc). Value types are stored directly in stack memory. All value types (have an initial value of zero (or, the zero equivalent of the type—for example, a Boolean's default value is zero, a date's default value is the minimum date value).

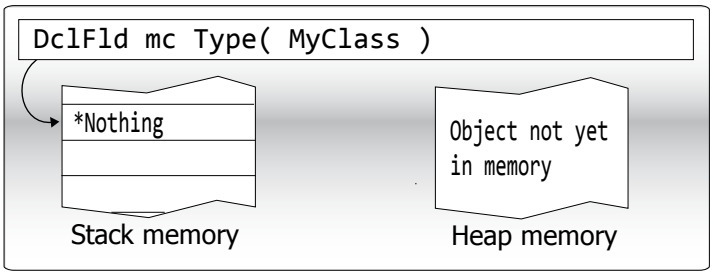


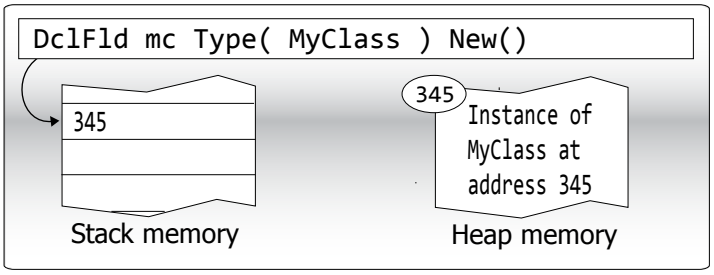
When a value type is declared, it is added to the stack with an initial value of zero (or zero's equivalent for the data type). Value types are not explicitly instanced.

Reference types

Reference types don't have a fixed size—objects vary in size from object to object (some are very large, some very small). Reference types have two parts: an instance variable that lives on the stack and (after instancing) the actual class instance that lives in the heap memory. All reference types (except for *Char) have an initial value of *Nothing (null). Until explicitly instanced reference types do not have a value. In this state, the value is known as null, or in AVR parlance, *Nothing.

When a reference type is instanced, its value (in this case, the variable *mc*), its value is the heap address where the object instance resides. Note that this value, this address, is not available to you directly. You can't use this address location to shoehorn in other code at that address (like you can with C++).





Reference types are not implicitly instanced (like value types are). Until you instance a reference type, its value is *Nothing (null).

Reference types can be instanced upon their declaration by adding the New() keyword to the declaration. They can also be instanced inline (when you want to defer the instancing of a class) using the *New operator as shown below:

```
DclFld mc Type( MyClass )
  .
  .
  .
mc = *New MyClass()
```

Working with reference types

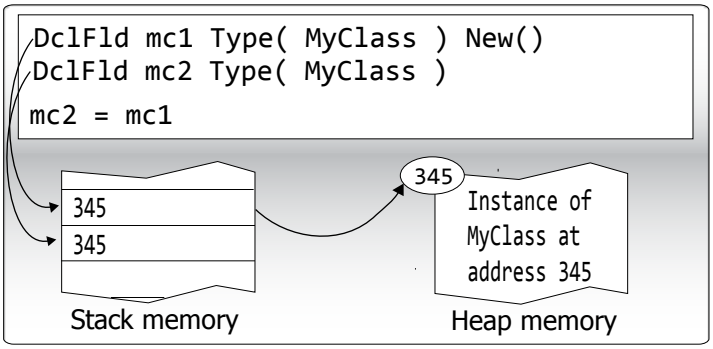
Given what we've just learned about value types and reference types, what do you think the output of this snippet is? Be careful!

```
DclFld mc1 Type( MyClass ) New()
DclFld mc2 Type( MyClass )

mc1.Balance = 23.45

mc2 = mc1

mc2.Balance = 77.98

MsgBox mc1.Balance
```

The ouput is often surprising to beginners. What is the output? Hint: how many instances of AccountPayment exist now?

The output is 77.98. Not 23.45. How can that be? The second assignment assigned a value to the *ap2* instance, not the *ap1* instance?

The issue to understand here is that when you assign reference types from one to another, you are simply assigning the memory reference. Thus, by assigning *ap1* to *ap2*, you're saying that *ap1* and *ap2* both reference the *same* object. In this case, there is only one instance of AccountPayment and both *ap1* and *ap2* reference that single instance. Therefore, through either reference you're changing the same object instance's Balance property.



Once the ap2 = ap1 assignement has been performed, both object instances reference the same object The picture to the left shows memory status after the ap2 = ap1 assignement.

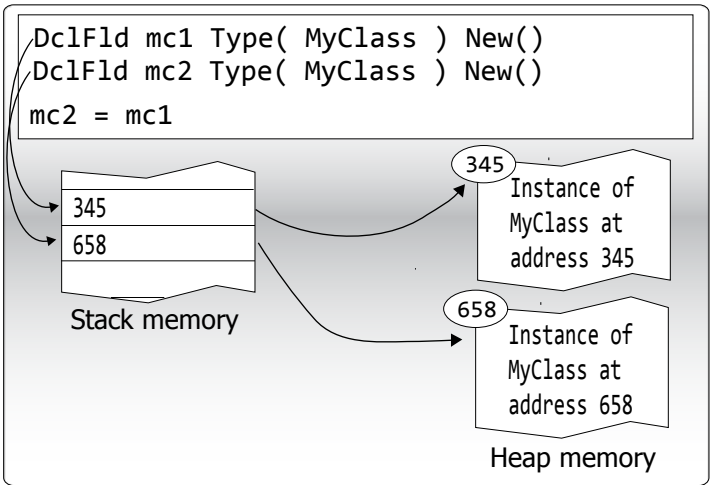
A lot of programmers will look upon this reference assignment behavior as though its a bad thing. It's not bad! It's a very good thing. The ability to pass references to objects from other objects is key to data encapsulation. Reference types provide a very effecient way to eliminate your dependence on global variables (and that's just one benefit—you'll see others later).

Throwing you a curve ball

Given what you've just learned, what is the output of the snippet below? Carefully notice the difference between this snippet and previous one. In this case, both mc1 and mc2 are instanced with the New() keyword. How does this difference change the output of the snippet?

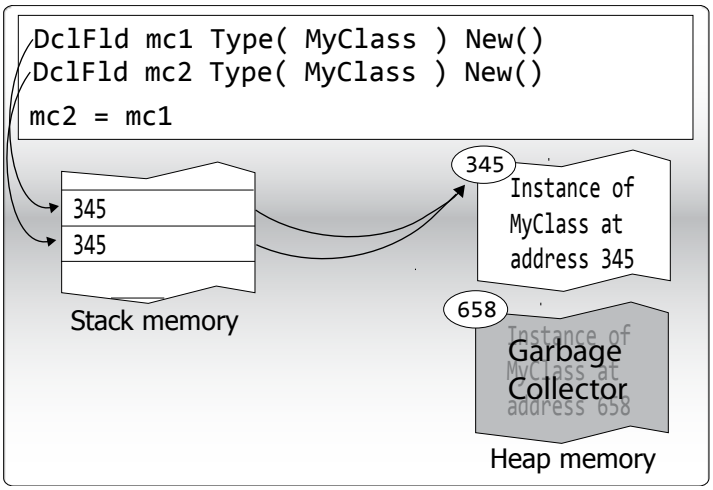
```
DclFld mc1 Type( MyClass ) New()
DclFld mc2 Type( MyClass ) New()

mc1.Balance = 23.45

mc2 = mc1

mc2.Balance = 77.98

MsgBox mc1.Balance
```

The output of snippet is 77.98. Exactly the same as the previous snippet! How can that be? The code above, as shown below clearly, creates two separate instances of MyClass.



Until the mc2 = mc1 line is performed, there are indeed two instances of MyClass created.

However, after the mc2 = mc1 assignment, bot class instances are once again referring to the same object in memory—in this case the one living at address 345.



After the mc2 = mc1 assignment, both variables are pointing to the same object in memory. The snippet output is the same as the previous exercise—it doesn't matter than mc2 was initially instanced.

Once mc2 has been reassigned to mc1, its original instance is relegated to the .NET garbage collector. Once processed by the garbage collector, that object's memory is reclaimed and available for use for something else.

The *Char and *String data types

AVR for .NET offers two data types to represent character strings: *String and *Char. The *Char type stores RPG fixed-length strings and *String stores variable-length strings. Both are based on .NET's intrinsic System.String data type, but the two have a significant differences: *Char variables are always set to the number of blanks as specified for the length of the field. *String variables do not have a default value and will be null until a value is assigned.

Both *Char and *String are reference types. However, unlike all other reference types, *Char has the special-case behavior of always be intialized to blanks upon its declaration. The *Char type is the *only* reference type to have an initial value upon its declaration.

Consider the code below:

```
DclFld y Type( *String )
DclFld x Type( *Char ) Len( 5 )
```

In this case, y doesn't have a value and x's value is five blanks.

It's important to remember that, by default, *String variables don't have an implicit value until you've assigned one. This behavior could cause problems. For example, the code below will fail with a runtime error because y doesn't yet have a value:

```
DclFld y Type( *String )
DclFld x Type( *Integer4 )

x = y.Length
```

You can change this default behavior with the Inz() keyword. The line of code below

```
DclFld y Type( *String ) Inz( "     " )
```

declares a string variable named y with an initial value of five blanks.

As a general rule, use *String for all work variables in your program. Once you get used to them, they are much more convenient than *Char variables (because you don't need to declare their length).

Whew!

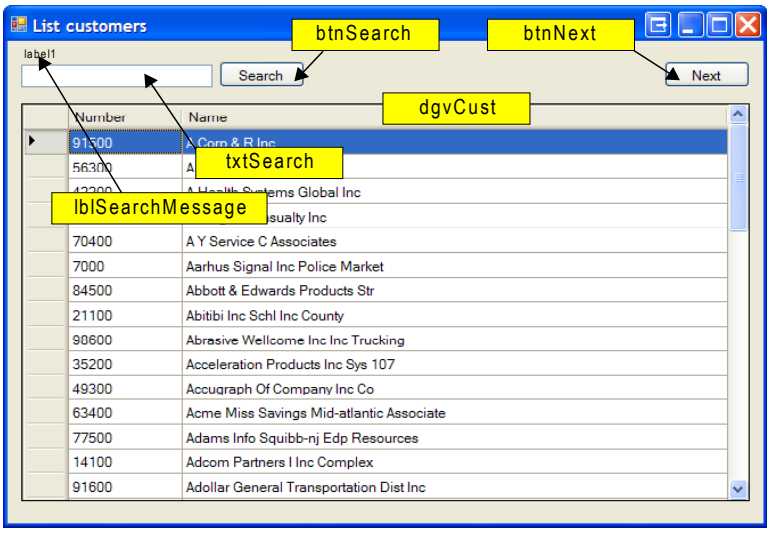
Confused? I hope not too much so. There is a lot to learning object oriented concepts and the absolute best thing you can do to start learning what was covered in this chapter is to write a few 20 line programs (it should never take more than that) to see this chapter's concepts in action. Roll up your sleeves, get your fingers dirty, and make friends with AVR's classes and OO capabilities. The quality of your enterprise applications depend on it!

Simple Windows List

The Simple Windows List introduces you to how to create a simple subfile-like navigational list in a Windows program. This exercise introduces several concepts, including:

- The MemoryFile and its Dataset
- Functions
- How to create and program a control-based user interface
- How to program with events, methods, and properties
- How to use program events to control program logics
- How use methods and properties to perform actions and manipulate control behavior

This document discusses the Windows Windows List project and the throught process driving its creation. After discussing briefly how to create the user interface, this document provides a detailed code narrative explaining all aspects of this project's source code.



Program intent: the purpose of this simple project is to introduce several important concepts of creating an AVR for .NET Windows program. In the real world, this program (or one like it) would server as a navigational front-end for users to select customers with which to perform an action. This simple program features two operations: a *Next* operation and a *Search* operation. The Next operation loads the grid with the next group of customers and the Search operation loads the grid with customers starting at a given customer name.

The Windows form this project uses needs only five controls:

1. A TextBox box for the search value. Name this control txtSearch.
2. A Button for the Search operation. Name this control btnSearch.
3. A Button for the Next operation. Name this control btnNext.
4. A DataGridView to display customer info in a grid. Name this control dgvCust.
5. A Label control to display an error message if the search fails.

Add these controls, assigning their names as you go, so that they are positioned to approximate the figure below (you can make placement and size adjustments as you go along). To see the Properties for a given control, select it ("give it focus") and press F4. More detail on setting other properties for these controls follows.

Setting the control properties

Control	Property	Value
txtSearch	Name	txtSearch
btnSearch	Name	btnSearch
lblSearchMessage	Name	lblSearchMessage
	Text	Label1
btnNext	Name	btnNext
	Text	Next
dgvCust	Name	dgvCust
	AllowUserToAddRows	False
	AllowUserToDeleteRows	False
	AllowUserToOrderColumns	False
	AllowUserToResizeColumns	False
	AllowUserToResizeRows	False
	Anchor	Top, Bottom, Left, Right
	ReadOnly	True
	SelectionMode	FullRowSelect
	MultiSelect	False

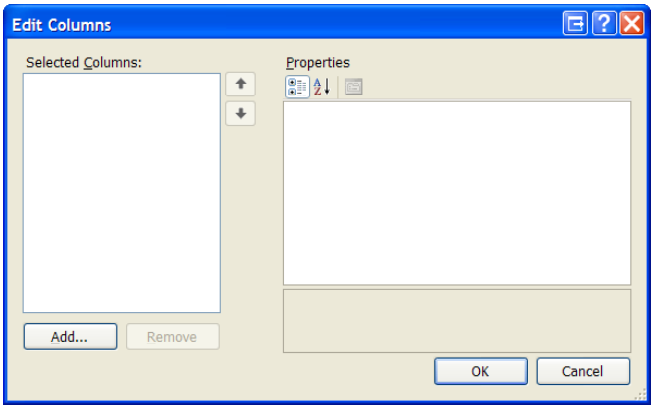
Notes:

The lblSearchMessage Text property defaults to Label1. Leave it at this default value. The program changes this property as needed at runtime. By default, this property needs to be blank. However, if you make it blank at design time, that makes it very hard to find the label during design.

The dgvCust DataGridView AllowUserTo... properties narrow the behavior of the grid for this program's specific purposes. Depending on you use grids in other programs, you'lkl probably want to experiement with this properties. The Anchor property is interesting, when set to "Top, Bottom, Left, Right," it lets the grid expand as the form size changes at runtime. As you're initially configuring the start-up size of the form and grid, you may find it helpful to set this property to "Top ,Left" only temporarily. Otherwise the grid grows with the size of the form even at design time. As you work with the Anchor property, get familar with its special custom property dialog (show when you click the dropd-own arrow in the Anchor property window). Setting ReadOnly does just what it sounds ilke it does; it makes the grid not input capable. And finally, because this grid is intended primarily for navigational purposes, setting SelectionMode to FullRowSelect ensures that a full row is always selected (as opposed to individual cells within a row).

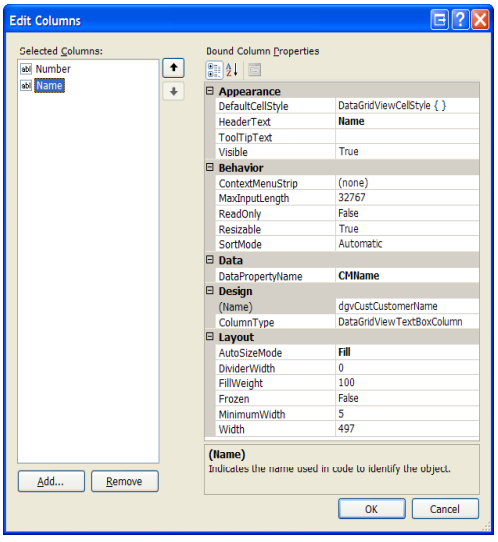
Add two columns to the dgvCust DataGridView

The dgvCust DataGridView needs two columns, one for the customer number and one for the customer name. From the design time view of ListForm.vr (see "Viewing a Windows form in design time" in the appendix) , click the dgvCust to give it focus. Press "F4." Click the "Columns" property (to give it focus) and then click the elipisis that shows on the right. This displays the DataGridView's "Edit Columns" dialog (as shown to the left). This dialog lets you add or remove columns to the DataGridView.



Click the "Add" button and add a DataGridView.TextBoxColumn column for the customer number. Name it dgvCustCustomerNumber and set its heading to "Number." Add a second DataGridView.TextBoxColumn and name it dgvCustCustomerName. Set its heading to "Name." After adding these two columns, you can then select them (one at a time) from the "Selected columns" list and further define other properties for each. The Number column needs no further work, but select the Name column and set its AutoSizeMode property (in the Layout section) to "Fill." When you're done, the two columns should have their properties set as shown in the table below.

Control	Property	Value
Number column	Name	dgvCustCustomerNumber
	DataPropertyName	CMCustNo
	HeaderText	Number
Name column	Name	dgvCustCustomerName
	DataPropertyName	CMName
	HeaderText	Name
	AutoSizeMode	Fill



This is what the properties Windows looks like for any one DataGridView column.

Program narrative

With the user interface created, here is a narrative of the source code for the Simple Windows List class exercise. This code narrative represents the finished project. It covers all of the concepts in detail that were discussed in class during the creation of this project.

The lines number correspond to both the source listing in Appendix A — Project source listing and the source of the downloadable example project. (To ensure that Visual Studio shows line numbers for source code, use the Tools>Text Editor>Basic menu option. In the display section on the right, check "Line Numbers."

Lines 1 - 7: Default Using statements for a Windows program. Namespaces specified by Using statements let you make unqualified references to classes in these namespaces.

Lines 10 - 23: Declarations for all the controls on the form. These were created implicitly by the Windows forms designer.

Line 25: Declare a DB object. This object is what connects your AVR program to its underlying database server. In AVR, the server platform can either an i5, SQL Server, or the ASNA local DB. In this case, we're using the database specified by the *Public/DG Net Local database name. This is the examples database that installs locally on your PC when you install AVR. Later in this exercise we'll see how to change this so that this program connects to an i5.

Lines 27 - 33: Declare a disk file. The file CMastNewL2, from the Examples library, is opened for indexed, read-only input. Its program name is CustByName, it must be explicitly opened, and uses a field prefix of "Cust_." This field prefix helps provide a pseudo fully-qualified file/field name in the program. See Appendix A for the layout of CMastNewL2's record format. CMastNewL2 is a logical file, keyed first on customer name (CMName) and then customer number (CMCustNo).

Lines 35 - 40: Declare a memory file. A memory file is an in-memory work file. In this case, the memory file has exactly the same layout, including field prefix, as the data file file declared in lines 28-34. Records are read from the data file and written to this memory file for the purpose of "binding" records written to the memory file to the DataGridView. Think of this memory file a bit like an externally described, multiple occurrence data structure. Memory files do truly live in memory, thus the more records you write to a memory file, the more memory you are taxing.

Line 43: Declare a global long integer variable named RecordsToRead. This variable controls how many records are loaded into the DataGridView at any one time. Note, don't confuse an *Integer4, which is a long integer with a maximum value of 2,147,483,647 with a Zoned 4,0 value. In *Integer4, the 4 represents bytes, not number of digits.

RecordsToRead is global—that is, the variable is scoped to the class rather than to a routine. Thus, it's referenced as **This.RecordsToRead* in the program. **This* indicates the variable is owned by the class, as opposed to being local to a subroutine or function. Prefixing global variables with *This fully qualify

Lines 47 - 48: Declare two global variables, one named LastCustomerName and the other named LastCustomerNumber. These variables are declared like the fields Cust_CMName and Cust_CMCustNo in the CMastNewL2's record format These variables will be used by the Next operation to display the next group of customer records.

```
0001  Using System
0002  Using System.Collections
0003  Using System.ComponentModel
0004  Using System.Data
0005  Using System.Drawing
0006  Using System.Text
0007  Using System.Windows.Forms
0008
0009  BegClass ListForm Extends(System.Windows.Forms.Form) Access(*Public)
0010      DclFld dgvCust System.Windows.Forms.DataGridView Access(*Private) +
0011             WithEvents(*Yes)
0012      DclFld btnSearch System.Windows.Forms.Button Access(*Private) +
0013             WithEvents(*Yes)
0014      DclFld txtSearch System.Windows.Forms.TextBox Access(*Private) +
0015             WithEvents(*Yes)
0016      DclFld btnNext System.Windows.Forms.Button Access(*Private) WithEvents(*Yes)
0017      DclFld lblSearchMessage System.Windows.Forms.Label Access(*Private) +
0018             WithEvents(*Yes)
0019      DclFld dgvCustCustomerNumber System.Windows.Forms.DataGridViewTextBoxColumn +
0020             Access(*Private) WithEvents(*Yes)
0021      DclFld dgvCustCustomerName System.Windows.Forms.DataGridViewTextBoxColumn +
0022             Access(*Private) WithEvents(*Yes)
0023      DclFld components Type(System.ComponentModel.IContainer) Inz(*Nothing)
0024
0025      DclDB pgmDB DBName( "*Public/DG Net Local" )
0026
0027      DclDiskFile  CustByName +
0028            Type( *Input  ) +
0029            Org( *Indexed ) +
0030            Prefix( Cust_ ) +
0031            File( "Examples/CMastNewL2" ) +
0032            DB( pgmDB ) +
0033            ImpOpen( *No )
0034
0035      DclMemoryFile CustMem +
0036            DBDesc( "*Public/DG Net Local" ) +
0037            Prefix( Cust_ ) +
0038            FileDesc( "Examples/CMastNewL2" ) +
0039            ImpOpen( *No ) +
0040            RnmFmt( CustMemR )
0041
0042      // Global variable that controls number of rows in grid.
0043      DclFld RecordsToRead Type( *Integer4 )
0044
0045      // Global variables to store last customer name and number
0046      // current in the grid.
0047      DclFld LastCustomerNumber Like( Cust_CMCustNo )
0048      DclFld LastCustomerName   Like( Cust_CMName )
0049
```

Lines 53 - 63: Subroutine called when the btnNext button is clicked. Clicking this button puts the next group of records in the dgvCust DataGridView.

- Line 58: Clear the MemoryFile's file data.
- Line 60: Use SetLL to position to the next customer in the list. (passing LastCustomerName and LastCustomerNumber as arguments). This logic carries over the customer on the last row as the customer on the first row of the next display. If you're rather not carry over this last name, use a SetGT here rather than a SetLL.
- Line 62: Call the LoadGrid() subroutine to put the next group of records in the dgvCust DataGridView.

Lines 65 - 71 Call the PerformCustomerSearch() subroutine to search for the customer name entered in the txtSearch textbox. Because PerformCustomerSearch() is used by both the Next and Search operations, it needs to receive both the customer name and number. For the Search operation, you should pass a zero for the customer number, letting the search find the first customer name requested.

Lines 73 - 79: Subroutine that handles a Windows form's Activated event. This event fires immediately after the form is loaded. It is a good place to intial form setup, such as setting focus to an initial control. In this case, whenever the form is activated, the dgvCust DataGridView is focused.

Lines 81 - 87: Subroutine called that handles just before the form closes (when the operator ends the program by clicking the form's X). This subroutine calls the CloseData() subroutine which closes data files and disconnects the program from the database server.

Lines 89 - 106: Subroutine called when the Windows form is intially loaded into memory. Conceptually, if you're familar with green-screen RPG's *INZSR subroutine, this routine is analgous (to a a degree) to that. This is a good place to put initial program housekeeping. For this program, when the form is loaded:

- Line 94: Clear the txtSearchMessage Text property.
- Line 96: Set records to read to 50.
- Line 99: Set the dgvCust's DataGridView AutoGenerateColumns property to false. This property ensures that only explicitly-created columns will be shown in the DataGridView. This property must be set manually! It cannot be set with the Properties window. Also, watch that it might not display in Intellisense. If it doesn't, don't worry about that; it is indeed there.
- Line 101: Call the Open() subroutine to connect to the database server and open files.
- Line 105: Call the LoadGrid() subroutine to put the initial group of records in the dgvCust DataGridView.

```
0050      //-------------------------------------------------------------------------
0051      // Event handlers.
0052      //-------------------------------------------------------------------------
0053      BegSr btnNext_Click Access(*Private) Event(*this.btnNext.Click)
0054          DclSrParm sender *Object
0055          DclSrParm e System.EventArgs
0056
0057          // Clear the memory file.
0058          CustMem.ClearFileData()
0059          // Position to next customer.
0060          SetLL CustByName Key( *This.LastCustomerName, *This.LastCustomerNumber )
0061          // Load grid with records.
0062          LoadGrid()
0063      EndSr
0064
0065      BegSr btnSearch_Click Access(*Private) Event(*this.btnSearch.Click)
0066          DclSrParm sender *Object
0067          DclSrParm e System.EventArgs
0068
0069          // Search for the customer name entered in the search textbox.
0070          PerformCustomerSearch( txtSearch.Text.Trim(), 0 )
0071      EndSr
0072
0073      BegSr ListForm_Activated Access(*Private) Event(*this.Activated)
0074          DclSrParm sender *Object
0075          DclSrParm e System.EventArgs
0076
0077          // Set initial focus to the DataGridView.
0078          dgvCust.Focus()
0079      EndSr
0080
0081      BegSr Form_FormClosing Access(*Private) Event(*this.FormClosing)
0082          DclSrParm sender Type(*Object)
0083          DclSrParm e Type(System.Windows.Forms.FormClosingEventArgs)
0084
0085          // Close files and DB connection.
0086          *This.Close()
0087      EndSr
0088
0089      BegSr Form1_Load Access(*Private) Event(*this.Load)
0090          DclSrParm sender *Object
0091          DclSrParm e System.EventArgs
0092
0093          // Clear search text.
0094          lblSearchMessage.Text = String.Empty
0095          // Set 50 records as grid "page size."
0096          *This.RecordsToRead = 50
0097          // Do let the DataGridView autogenerate any columns other
0098          // than those explicitly created.
0099          dgvCust.AutoGenerateColumns = *False
0100          // Open DB connection and files.
0101          *This.Open()
0102          // Load grid with initial rows. At this point, the file will always
0103          // be at beginning-of-file, so the initial rows are always from the
0104          // beginning of the file.
0105          LoadGrid()
0106      EndSr
0107
```

This is a very important line of code! This is the only way to stop the DataGridView from automatically generating columns..

Lines 111 - 118: Close() subroutine that closes the files and disconnects the database server connection. It is very important to close files and disconnect the database server connection before the program ends. Without doing so, you'll files unnecessarily opened and have a potential orphan database job running (especially on the i5). For Windows applications, the FormClosing event handler is a good place to call Close().

From a performance point of view, when used with connection pooling, disconnecting the DB server will actually improve performance. A DB server job isn't put back in the available pool of jobs until it has been disconnected. We'll discuss this topic more when we discuss connection pooling.

Note that both the Open() and Close() methods are prefixed with *This to avoid confusion for the compiler between calling these methods and AVR's Open and Close operation code.

Lines 120 - 142: LoadGrid() subroutine that read x records from the data file and loads them into the dgvCust DataGridView. To actually get the records into the dgvCust DataGridView, LoadGrid() writes them to the CustMem file. The dgvCust DataGridView ultimately uses the CustMem's DataSet property as its data source.

Lines 123 -129: A do loop that reads as many records as specified by the global RecordsToRead variable from data file CustByName and writes them to memory file CustMem. Because both files are defined with the same underlying database file, including the field prefix, no manually assignment of field values is necessary here; when a record is read from the data file Cust, the buffer for memory file Cust is implicitly field. If EOF is countered during the read loop, the loop is exited.
Note that to determine EOF, the data file Cust's IsEof property is used . The test for EOF could have also been done with the %EOF() BIF like this:

```
If ( %EOF( CustByName )
.
EndIf
```

Lines 132 - 133: Save the last customer name and number read to global variables LastCustomerName and LastCustomerNumber. These values will be used later for the Next operation. These two lines show how readable field names are when used with a good prefix (in this case, the prefix being "Cust_."

Line 137: Assign the dgvCust DataGridView's DataSource property. This is the line that actually populates the grid with data in the memory file.

Line 140: Use the results of the MoreRecordsForward() function to toggle btnNext's Enabled property.

Lines 144 - 154: MoreRecordsForward() function that returns true if there are more records to read foward from the last customer currently showing in the grid; otherwise this function returns false. This function works by using SetGT to attempt to position the file pointer beyond the customer name and number specified. If a record was found, true is returned; otherswise false is returned.

Lines 156 - 163: OpenData() subroutine that connects to the database server and opens files. In the real world, you'll want a little more error handling than is shown here (a little more than the *none* shown here!). We'll discuss robust error handling later.

```
0108      //------------------------------------------------------------------------
0109      // Private subroutines and functions.
0110      //------------------------------------------------------------------------
0111      BegSr Close
0112          // Close CustByName file.
0113          Close CustByName
0114          // Close CustMem file.
0115          Close CustMem
0116          // Disconnect DB connection.
0117          Disconnect pgmDB
0118      EndSr
0119
0120      BegSr LoadGrid
0121          // Starting at current file position, read next x records
0122          // and write them to the memory file.
0123          Do FromVal( 1 ) ToVal( *This.RecordsToRead )
0124              Read CustByName
0125              If ( CustByName.IsEof )
0126                  Leave
0127              EndIf
0128              Write CustMem
0129          EndDo
0130
0131          // Save last customer name and number displayed.
0132          *This.LastCustomerName   = Cust_CMName
0133          *This.LastCustomerNumber = Cust_CMCustNo
0134
0135          // Set zeroth table in the CustMem's DataSet as the
0136          // datasource.
0137          dgvCust.DataSource = CustMem.GetFileData()
0138
0139          // Should the Next button be enabled?
0140          btnNext.Enabled = MoreRecordsForward( *This.LastCustomerName, +
0141                                                *This.LastCustomerNumber )
0142      EndSr
0143
0144      BegFunc MoreRecordsForward Type( *Boolean )
0145          DclSrParm CustomerName   Like( Cust_CMName )
0146          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0147
0148          // Determine if there are more records to read after
0149          // the last record currently displayed in the grid.
0150
0151          // Attempt to read beyond last customer read.
0152          SetGT CustByName Key( CustomerName, CustomerNumber )
0153          LeaveSr CustByName.IsFound
0154      EndFunc
0155
0156      BegSr Open
0157          // Open DB connection.
0158          Connect pgmDB
0159          // Open CustByName file.
0160          Open CustByName
0161          // Open MemoryFile.
0162          Open CustMem
0163      EndSr
0164
```

Lines 165 - 189 Subroutine called when the grid needs to be refilled starting from a given customer number and name. The pseudo code for this routine is:

```
if a customer record is found with the name and number requested
    clear the MemoryFile
    load the grid starting at the customer for which you're searching
    clear the customer search name
    set focus to the dgvCust DataGridView
else
    set the search message to "Search failed."
    set focus to the customer search name
endif
```

The ASNA test data has a case-sensitive index, so the value to search for is case-sensitive (ie, to search for *SMITH* type in *Smith).*

Line 71 makes a call to the SearchForCustomer() function, passing it the customer name provided in txtSearch.Text and zero (thus searching for the first customer name matching the search criteria). The call to SearchForCustomer() is written like this:

```
If ( SearchForCustomer( txtSearch.Text.Trim(), 0 ) )
```

but could have also been written like this:

```
If ( SearchForCustomer( txtSearch.Text.Trim(), 0 ) = *True )
```

the = **True* is generally considered redundant and isn't usually provided. However, especially for beginning AVR coders, its presence may be helpful. Having said that, do learn to read the code without the = **True*, you will see it coded more that that way.

It may seem unnecessary to pass the customer number for this search. However, you'll soon see why it is necessary (hint, SearchForCustomer is also going to be a helper for the Next operation).

If the customer searched for was found:

Line 171: Attempt to position the file to the name and number provided.
Line 172: If the file positioning was successful...
Line 174: Clear the CustMem memory file.
Line 176: Call the LoadGrid() subroutine to put the next group of records in the dgvCust DataGridView.
Line 178: Clear txtSearch's Text property by setting its value to String.Empty. String.Empty is .NET-provided synonym (think of it as a constant) for an empty string. Using String.Empty is the functional equivalent of:

```
txtSearch.Text = ''
```

but with the advantage of being a little more clear and explicit (and, if you are a cycle-counter, using String.Empty is a bit more effecient. More on why much later!
Line 180: Clear lblSearch's Text property.
Line 182: Set focus to the dgvCust DataGridView.

If the customer searched for was not found:

Line 185: Set the search message to "Search failed."
Line 187: Position the cursor in the txtSearch textbox.

Lines 191 - 193: These lines are abridged here, but in the example source these lines are where the constructor is, where InitializeComponent() is (where all of the form controls are all declared) and where Dispose() is. This code is all created automatically by Visual Studio's Windows form designer. You'll rarely, if ever, need to add or modify code to any of these three areas. For nearly all programs, you will leave these routines exactly as the Windows designer generates therm.

```
0165      BegSr PerformCustomerSearch
0166          DclSrParm CustomerName   Like( Cust_CMName )
0167          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0168
0169          // If the search was successful...
0170          // Set lower limits at name and number provided.
0171          SetLL CustByName Key( CustomerName, CustomerNumber )
0172          If ( CustByName.IsFound )
0173              // Clear existing rows from grid.
0174              CustMem.ClearFileData()
0175              // Load the grid with records.
0176              LoadGrid()
0177              // Clear search value.
0178              txtSearch.Text = String.Empty
0179              // Clear search message.
0180              lblSearchMessage.Text = String.Empty
0181              // Set focus to the DataGridView.
0182              dgvCust.Focus()
0183          Else
0184              // Set search message.
0185              lblSearchMessage.Text = “Search failed.”
0186              // Position cursor at search text.
0187              txtSearch.Focus()
0188          EndIf
0189      EndSr
0190
0191      //
0192      // This abridged listing has omitted the Windows forms designer-generated code.
0193      //
0194  EndClass
```

This page intentionally left blank.

Simple Windows list with an update panel

The AddUpdate form lets the user change field values for a given customer. The update form (shown below) is displayed when a user double-clicks a row in the customer list from the previous exercise.

Program intent: the purpose of this simple project is to introduce some of the important concepts of navigating from one from to another and using AVR file IO operations to write changes to disk.

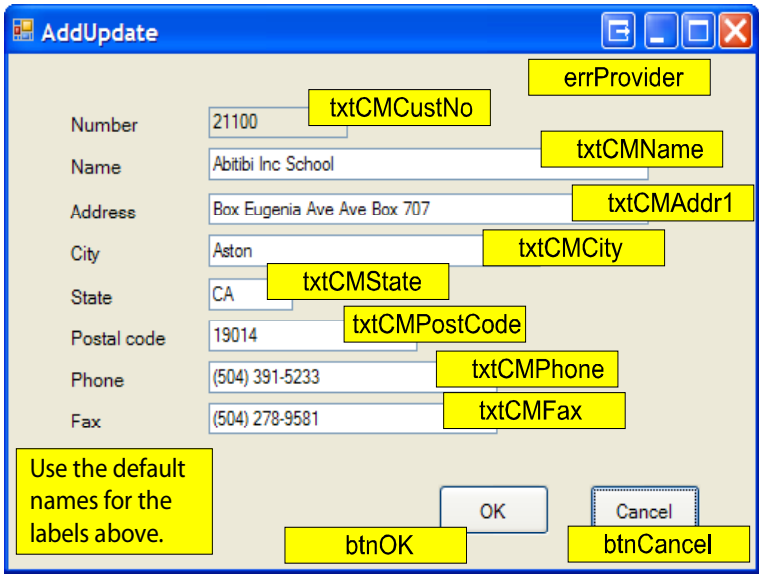
This part of the program illustrates how to:

- Do a database update with AVR
- Implement the SharedDB pattern.
- Open a form from a another form, passing information back and forth
- Perform input validation
- Lay the groundwork for optimistic record locking. This version of the program doesn't fully implement optimistic record locking, but it doesn't lock the record until it's ready for update. Another example will show you the full details on optimistic locking.

To build this program, copy the SimpleCustomerList project to another folder. Add a Windows form named AddUpdate. The AddUpdate form in this project needs these controls:

1. A TextBox for the customer number. Name this control txtCMCustNo
2. A TextBox for the customer name. Name this control txtCMName.
3. A TextBox for the customer address Name this control txtCMAddr1.
4. A TextBox for the customer city. Name this control txtCMCity.
5. A TextBox for the customer state. Name this control txtCMState.
6. A TextBox for the customer postal code. Name this control txtCMPostCode.
7. A TextBox for the customer phone. Name this control txtCMPhone.
8. A TextBox for the customer fax. Name this control txtCMFax.
9. An ErrorProvider to display error messages. Name this control errProvider.
10. An OK button. Name this control btnOK.
11. A cancel button. Name this control btnCancel.

Also, add a Label control to the left of each of the eight TextBoxes. Use the default name for this controls, but assign their Text property as shown below. When you're done, the form should look similar to the one below.



Setting the control properties

In addition to the control names (and the eight labels' Text properties) set these controls' properties as show:

Control	Property	Value
AddUpdate	FormBorderStyle	FIxedSingle
	AcceptButton	btnOK
	CancelButton	btnCancel
	StartPosition	CenterParent
	Text	AddUpdate
txtCMCustNo	TabStop	False
txtCMName	TabStop	0
txtCMAddr1	TabStop	1
txtCMCity	TabStop	2
txtCMState	TabStop	3
txtCMPostCode	TabStop	4
txtCMPhone	TabStop	5
txtCMFax	TabStop	6
btnOK	Text	&OK
btnCancel	Text	&Cancel
errProvider	BlinkStyle	NeverBlink

The AcceptButton and CancelButton properties on the AddUpdate form causes the OK button be the default action on the form when the Enter key is pressed; the CancelButton property causes the Cancel button to be the default action on the form when the Escape key is pressed.

In the Text property of the two buttons, you'll notice both start with an ampersand (&). This assigns implicit speed keys to these two buttons. Alt-O is the same as clicking the OK button and Alt-C is the same as clicking the Cancel button.

This page is intentionally left blank.

Program narrative for ListForm.vr

The ListForm.vr user interface doesn't change in this example. However, you do need to add code to it in three places. These additions are shown on the facing page. The line numbers in the code to the right are for referencing the lines of code to change or add—they *do not* show where to add these lines in ListForm.vr. See the the narrative below for details on these changes.

Lines 1 and 2: Declare and instance the AddUpdate form. Add these two lines immediately between the CustMem memory file declaration and the RecordToRead variable declaration in the class mainline. Because the AddUpdate form is instanced on this line, this means it is available for use the duration of the life of the ListForm form. For frequently used forms, this is typically what you'll do; declare it globally and instance it immediately. For less frequently used forms, it may be advantagous to instance them as needed and then let them go out of scope until needed again.

Lines 4 - 20: Handle the double-click on a row on the DataGridView. This routine lets a user select a row by double-clicking on it. When a row is selected, the AddUpdate form is displayed for the customer in the row selected.

- Line 8: Declare a variable named dgvr of type DataGridView row. This variable is used to get a reference to the DataGridView row the user double-clicked.
- Line 9: Declare a variable named RowNumber of type *Integer4. This variable tracks the zero-based row number the user double-clicked.
- Line 12: Get a reference to the selected row. Because the DataGridView's MultiSelect property is false, the user can only select one row. In this case, then, getting the selected row simply requires asking for the zeroth row of selected rows.
- Line 14: Assign the value in the "dgvCustCustomerNumber" column as the customer number selected (recall that column names were assigned when you added columns to the DataGridView). This code has a potential flaw in that the logic is dependent on the zeroth column always being the customer number. Should another coder shuffle columns around in the DataGridView, your code is going to have columns here. Beware!
- Line 16: Call the UpdateCustomer() function in the AddUpdateForm, passing it the customer number selected. If this function returns true, the update operation succeeded; otherwise it did not. This function is what indirectly causes the AddUpdateForm to display.
- Line 17: If the update operation succeeded, reload the DataGridView using the customer number and (potentially) updated customer name. The AddUpdate form provides two public variables that represent the customer number being updated and what, if any, changes were made to the customer name. The updated customer is always positioned first in the list (this way is the customer name is changed so that it moves out of alphebetic order in the list, the customer just updated is still shown in the grid).
  If you were to modify this code to allow adding a new customer, you'll probably also want the new customer added at the top of the list; thus no coding changes would be required here should an Add operation be needed.

Lines 26 - 28 Add the three lines shown to the Form_FormClosing event handler. Line 28 calls the AddUpdate form's Clos) method to close its files and disconnect its database connection. (Most of the forms you create should have a public routine like this to close files and disconnect its DB before the form goes out of scope.) Line 29 calls the AddUpdate form's Dispose() method to ensure its resources are appropriately garbage collected by .NET.

If you're thinking that these two lines might be easy lines to forget, you are correct. However, the tradeoff for these two lines is that once files are opened in the AddUpdate form, they stay opened for the duration of the program—adding substantially to performance.

```
0001      // Declare and instance the AddUpdate form.
0002      DclFld AddUpdateForm Type( AddUpdate ) New( pgmDB )
0003
0004      BegSr dgvCust_DoubleClick Access(*Private) Event(*this.dgvCust.DoubleClick)
0005          DclSrParm sender *Object
0006          DclSrParm e System.EventArgs
0007
0008          DclFld dgvr      Type( DataGridViewRow )
0009          DclFld RowNumber Type( *Integer4 )
0010
0011          // Get a reference to the selected row.
0012          dgvr = dgvCust.SelectedRows[ 0 ]
0013          // Fetch customer number column 0 in DataGridView.
0014          Cust_CMCustNo = dgvr.Cells[ “dgvCustCustomerNumber” ].Value.ToString()
0015          // If successful calling AddUpdateForm.UpdateCustomer()...
0016          If ( AddUpdateForm.UpdateCustomer( Cust_CMCustNo ) = *True )
0017              PerformCustomerSearch( AddUpdateForm.CustomerName, +
0018                                     AddUpdateForm.CustomerNumber )
0019          EndIf
0020      EndSr
0021
0022      BegSr Form_FormClosing Access(*Private) Event(*this.FormClosing)
0023          DclSrParm sender Type(*Object)
0024          DclSrParm e Type(System.Windows.Forms.FormClosingEventArgs)
0025
0026          // Close files and DB in AddUpdateForm.
0027          AddUpdateForm.Close()
0028          AddUpdateForm.Dispose()
0029          // Close files and DB connection.
0030          *This.Close()
0031      EndSr
```

Program narrative for AddUpdate.vr

Lines 1 -7: Default Using statements for a Windows program. Namespaces specified by Using statements let you make unqualified references to classes in these namespaces.

Lines 8: Add a Using statement for the System.Text.RegularExpressions namespace. Later in the code (line 158), the Regex.Replace() method is used to remove all non-numeric characters from a string.

Lines 11 - 30: Declarations for all the controls on the form. These were created implicitly by the Windows forms designer.

Line 32: Declare a DB object.

Lines 34 - 41: Declare a disk file. In this case, the CMastNewL1 file, from the Examples library, is opened for index, update and add access. See appendix A for CMastNewL1's record layout. The CMastNewL1 is a logical file keyed on customer number.

Lines 46-47: Two public variables to "publish" the customer number and (potentially updated) customer name. These values are for a calling form to know the number and and name of the changed customer (usually for the purpose of repositioning a selection list).

Line 50: A private Boolean value that indicates if the update operation was completed or not. It is completed if the user clicked OK and the record was successfully updated. This global value is used to set the return value of the UpdateCustomer() function.

Lines 55 - 62: Subroutine called when the Cancel button is clicked. If Cancel is clicked, the form is hidden (hiding a form removes if from view but keeps in it memory) and the global value UpdateComplete is set to false. After this event handler has been performed, control returns to line 181 in the UpdateCustomer() function.

```
0001  Using System
0002  Using System.Collections
0003  Using System.ComponentModel
0004  Using System.Data
0005  Using System.Drawing
0006  Using System.Text
0007  Using System.Windows.Forms
0008  Using System.Text.RegularExpressions
0009
0010  BegClass AddUpdate Extends(System.Windows.Forms.Form) Access(*Public)
0011      DclFld txtCMFax System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0012      DclFld label8 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0013      DclFld txtCMPhone System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0014      DclFld label7 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0015      DclFld txtCMPostCode System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0016      DclFld label6 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0017      DclFld label5 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0018      DclFld txtCMCity System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0019      DclFld label4 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0020      DclFld txtCMAddr1 System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0021      DclFld label3 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0022      DclFld txtCMName System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0023      DclFld label55 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0024      DclFld label1 System.Windows.Forms.Label Access(*Private) WithEvents(*Yes)
0025      DclFld btnOK System.Windows.Forms.Button Access(*Private) WithEvents(*Yes)
0026      DclFld btnCancel System.Windows.Forms.Button Access(*Private) WithEvents(*Yes)
0027      DclFld txtCMCustNo System.Windows.Forms.TextBox Access(*Public) WithEvents(*Yes)
0028      DclFld txtCMState System.Windows.Forms.TextBox Access(*Private) WithEvents(*Yes)
0029      DclFld errProvider System.Windows.Forms.ErrorProvider Access(*Private) WithEvents(*Yes)
0030      DclFld components Type(System.ComponentModel.IContainer) Inz(*Nothing)
0031
0032      DclDB pgmDB DBName( "*Public/Cherry5080" )
0033
0034      DclDiskFile  CustByNumber +
0035            Type( *Update ) +
0036            Org( *Indexed ) +
0037            Prefix( Cust_ ) +
0038            File( "Examples/CMastNewL1" ) +
0039            DB( pgmDB ) +
0040            ImpOpen( *No ) +
0041            AddRec( *Yes )
0042
0043      // These two public values are used to provide a calling form with the
0044      // customer number and (potentially updated) customer name when the
0045      // update operation succeeds.
0046      DclFld CustomerNumber Like( Cust_CMCustNo ) Access( *Public )
0047      DclFld CustomerName   Like( Cust_CMName )   Access( *Public )
0048
0049      // Did the update operation complete?
0050      DclFld UpdateComplete   Type( *Boolean )
0051
0052      //-------------------------------------------------------------------------
0053      // Event handlers.
0054      //-------------------------------------------------------------------------
0055      BegSr btnCancel_Click Access(*Private) Event(*this.btnCancel.Click)
0056          DclSrParm sender *Object
0057          DclSrParm e System.EventArgs
0058
0059          // Hide this form and set UpdateComplete to false.
0060          *This.Hide()
0061          *This.UpdateComplete = *False
0062      EndSr
0063
```

Lines 64 - 83: Subroutine called when the OK button is clicked. Clicking this button completes the update operation. After this event handler has been performed, control returns to line 181 in the UpdateCustomer() function.

Line 69:	Call the ValidateForm() Boolean function to see if the values entered pass data validation tests. If the data validates, ValiddateForm() returns true. More on the ValidateForm() in a moment.
Line 71:	If the data validated, call the UpdateRecord function passing the customer number being updated. Note that UpdateRecord is a Boolean function that, in this iteration of this program, unconditionally returns true. When error handling is added, the value of UpdateRecord() would be checked to make sure the record was updated successfully.
Lines 74 - 75:	Set the global values CustomerName and CustomerNumbers with values from the record buffer (this would reflect an updated customer name). Note that it is a good convention to prefix global class variables with *This. This disambiguates where the variable was declared (anytime you see a variable prefixed with *This, that variable *has* to have been declared globally).
Line 78:	Set the global value UpdateComplete to true.
Line 81:	Hide this form.

Lines 88 - 92: Clear all possible errors registered with the errProvider control. More on this in just a bit.

Lines 94 - 100: A public subroutine that closes the CustByNumber file and disconnects the DB connection. This routine is public and is intended to be called by the owning class (the class that instanced and showed the AddUpdate form). It is very important that this routine be called before the AddUpdate form goes out of scope.

Lines 102 - 111: A private subroutine to connect to the DB server and open the CustByNumber file. This subroutine gets called unconditionally each time the AddUpdate form is displayed. However, once the DB is connected and the files are opened they stay connected and opened until the CloseData() routine is called. Before any action is performed in OpenData() if first make sure that it needs to perform the connect and file open.

Lines 113 - 122: A private subroutine to populate CustByNumber's record format from the data collected with the user interface elements. When assigning character values from Textboxes, all that's needed is a simple assignment from the control's Text property to the field name. Other controls offer other similar properties to the Textbox's Text property. For example, a ComboBox has a SelectedValue property and a CheckBox has a CheckedProperty. For other field types, depending on the control holding the value, data conversion may be necessary before the assignment.

For example, at line 120, Cust_CMFax is assigned a value. Cust_CMFax is a packed, 10, 0 value. Therefore, the value from the txtCMFax Text property must have all non-numeric values removed before assigning the value to Cust_CMFax. The RemoveNonNumericCharacters() function (at lines 151-159) is used for this purpose. We'll discuss RemoveNonNumericCharacters() in more detail in a bit.

Values are trimmed to remove any spurious leading blanks.

```
0064      BegSr btnOK_Click Access(*Private) Event(*this.btnOK.Click)
0065          DclSrParm sender *Object
0066          DclSrParm e System.EventArgs
0067
0068          // If form data validates...
0069          If ( ValidateForm() )
0070              // Update record.
0071              UpdateRecord( Cust_CMCustNo )
0072
0073              // Set customer name and number for export.
0074              *This.CustomerName   = Cust_CMName
0075              *This.CustomerNumber = Cust_CMCustNo
0076
0077              // Set UpdateComplete true.
0078              *This.UpdateComplete = *True
0079              // Hide this form (which returns control to immediately after
0080              // the ShowDialog() statement in UpdateCustomer().
0081              *This.Hide()
0082          EndIf
0083      EndSr
0084
0085      //------------------------------------------------------------------------
0086      // Subroutines and functions.
0087      //------------------------------------------------------------------------
0088      BegSr ClearErrors
0089          // Clear all errors.
0090          errProvider.SetError( txtCMName,  String.Empty )
0091          errProvider.SetError( txtCMAddr1, String.Empty )
0092      EndSr
0093
0094      BegSr Close Access( *Public )
0095          // Close CustByNumber.
0096          Close CustByNumber
0097
0098          // Disconnect DB connection.
0099          Disconnect pgmDB
0100      EndSr
0101
0102      BegSr Open Access( *Public )
0103          // Open DB connection.
0104          If ( NOT pgmDB.IsOpen )
0105              Connect pgmDB
0106          EndIf
0107          // Open Cust file.
0108          If ( NOT CustByNumber.IsOpen )
0109              Open CustByNumber
0110          EndIf
0111      EndSr
0112
0113      BegSr PopulateRecordFromUI
0114          // Populate the record format from the user interface.
0115          Cust_CMName     = txtCMName.Text.Trim()
0116          Cust_CMAddr1    = txtCMAddr1.Text.Trim()
0117          Cust_CMCity     = txtCMCity.Text.Trim()
0118          Cust_CMState    = txtCMState.Text.Trim()
0119          Cust_CMPostCode = txtCMPostCode.Text.Trim()
0120          Cust_CMFax      = RemoveNonNumericCharacters( txtCMFax.Text )
0121          Cust_CMPhone    = txtCMPhone.Text.Trim()
0122      EndSr
0123
```

Lines 124 - 133: A private subroutine to populate the user interface elements from CustByNumber's most recently-read record format. When assigning character values to control properties, you'll generally want to Trim() them (remove trailing blanks). If you don't, the trailing blanks will be a part of the control value—and this is often off-putting to a user (for example, she'll need to backspace through the blanks to change the last character of a field).

For other data types, formatting may be necessary. Line 131 shows that the custom numeric formatting capabilities of the ToString() are used to format the fax number from its intrinsic 10,0 numeric value to a more pleasing representation of a fax number. If you'd rather not use ToString() to format a value, you can also use RPG edit codes and edit words. For example, you could use this:

```
txtCMFax.Text   = %EDITW( Cust_CMFax, "0(   )&    -    " )
```

to format the fax number appropriately. Having said that, once you get familar with ToString()'s formatting capabilities, you'll find that an intuitive, easy way to format values. You can read more detail on ToString()'s custom numeric formatting capabilities as this link:

http://msdn.microsoft.com/library/?url=/library/en-us/cpguide/html/cpconcustomnumericformatstrings.asp

While we're on the subject of formatting values, if you need to format a date data type, ToString() offers powerful date formatting as well. Read about it at the link below:

http://msdn.microsoft.com/library/?url=/library/en-us/cpguide/html/cpconcustomdatetimeformatstrings.asp

Lines 135 - 149: A function to read a record. Note that this function is very specifically for reading the record to display its values to the user (done with PopulateUIFromRecord() after the record is read). This routine *does not* lock the record. When its time to update the record, the record will be read again for update purposes. A later exercise discusses how to manage optimistic record locking.

Note that ReadRecord() is a function. In this exercise it unconditionally returns true. Later, when error handling is hooked up, it won't be hardwired to return true.

Lines 151 - 159: A function to remove non-numeric characters from a string. This function uses a regular expression to strip non-numeric characters from a given string. Regular expressions offer a succinct way to perform pattern matching and character replacement on string values. In this case, the line:

```
LeaveSr Regex.Replace( Value, "[^0-9]", String.Empty )
```

uses the Replace() method to search Value for any character matching the regular expression [^0-9] and replaces any found with String.Empty (a .NET intrinsic value for ""). [^0-9] is the regular expression that matches any occurrence of a non-numeric character.

If this class didn't have a

```
Using System.Text.RegularExpressions
```

at the top of its code, we couldn't have made the shorthand call to RegEx.Replace(), we would have had to make a fully qualified call like this:

```
LeaveSr System.Text.REgularExpressions.Regex.Replace( Value, "[^0-9]",
String.Empty )
```

As you can probably imagine, there is a lot to regular expressions. They will be covered in more depth later.

```
0124      BegSr PopulateUIFromRecord
0125          // Populate the user interface from the record format.
0126          txtCMName.Text     = Cust_CMName.Trim()
0127          txtCMAddr1.Text    = Cust_CMAddr1.Trim()
0128          txtCMCity.Text     = Cust_CMCity.Trim()
0129          txtCMState.Text    = Cust_CMState.Trim()
0130          txtCMPostCode.Text = Cust_CMPostCode.Trim()
0131          txtCMFax.Text      = Cust_CMFax.ToString( “(000) 000-0000” )
0132          txtCMPhone.Text    = Cust_CMPhone.Trim()
0133      EndSr
0134
0135      BegFunc ReadRecord Type( *Boolean )
0136          // Read the customer record for display.
0137          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0138
0139          // Read the record--do not lock the record.
0140          Chain CustByNumber Key( CustomerNumber ) Access( *NoLock )
0141          If ( CustByNumber.IsFound )
0142              // Populate the UI if record is found.
0143              PopulateUIFromRecord()
0144          Else
0145              // Error handling code here.
0146          EndIf
0147          // Assume the best for now!
0148          LeaveSr *True
0149      EndFunc
0150
0151      BegFunc RemoveNonNumericCharacters Type( *String )
0152          // Use a regular expression to remove all non-numeric characters
0153          // from an input string.
0154          DclSrParm Value Type( *String )
0155
0156          // [^0-9] is the regular expression that searches a string
0157          // for all occurrence of non-numeric values.
0158          LeaveSr Regex.Replace( Value, “[^0-9]”, String.Empty )
0159      EndFunc
0160
```

Lines 161 - 182: This public function is called by the consuming class to initiate updating a customer. This is an interesting function in that it is responsible for actually showing the AddUpdate form. Think about that—a form's own function can show its owning form. Before we investigate the workings of UpdateCustomer(), let's consider how it is called by a consumer. In the code below, the AddUpdate form is instanced as variable AddUpdateForm. That instance variable's UpdateCustomer() function is called to initiate the customer update:

```
If ( AddUpdateForm.UpdateCustomer( Cust_CMCustNo ) )
    // Do something here if the update was completed successfully.
EndIf
```

Ponder this code for a moment. Notice that the form that owns the AddUpdateForm instance variable isn't the form that shows the AddUpdate form! Rather, inside UpdateCustomer, the AddUpdate form shows itself (with the code *This.ShowDialog()). You'll often see consumer code like this:

```
AddUpdateForm.ShowDialog()
    // Do something after the form has been displayed.
EndIf
```

However, in the case above, there isn't a direct way for the AddUpdate form to convey to its caller if the update process finished successfully. Having a form responsible for showing itself has several advantages:

- By calling a function such as UpdateCustomer() a value can be directly returned to the call to indicate the status of the operation at hand.
- The AddUpdate form can be easily used by any form just by calling the UpdateCustomer() function. Consuming forms don't need to know other instructions for determining what comes after the ShowDialog() or Show(). For beginning programs, part of the power of this techique is easy to miss right, but later you'll appreciate that with this technique, the AddUpdate form doesn't know or care about what form is using it (other techniques for showing forms often brew a hard-to-decouple dependency between the parent and child form).

Line 165: UpdateCustomer() receives the customer number to update through the CustomerNumber parameteter. This line of code assigns that value to the txtCMCustNo Textbox. This TextBox is read-only.

Line 166: Set the initial focus to the txtCMName (customer name) TextBox.

Line 169: Connect to the DB server and open file(s). Although OpenData() is called uncondtionally, it provides the code needed to ensure a DB connect and file open(s) are really necessary.

Line 172: Calling ClearErrors() removes all previously registered errors from the errProvider ErrorProvider control. This ensure that there aren't any leftover errors if the previous display of the AddUpdate form ended by clicking the Cancel button when errors were present.

Line 175: Call ReadRecord() to read the customer record with the customer number specified. ReadRecord() also refreshes the user interface with values read from the customer record.

Line 178: Show the AddUpdate form modally. Being shown modally, the user can't return to the calling form until completing the use of the AddUpdate form.

Line 181: Leave the function with the value of the global variable UpdateComplete.

```
0161      BegFunc UpdateCustomer Type( *Boolean ) Access( *Public )
0162          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0163
0164          // Assign the customer number and set focus.
0165          txtCMCustNo.Text = CustomerNumber.ToString()
0166          txtCMName.Focus()
0167
0168          // Connect DB and open files.
0169          *This.Open()
0170
0171          // Clear all validation errors.
0172          ClearErrors()
0173
0174          // Read the record.
0175          ReadRecord( CustomerNumber )
0176
0177          // Show this form.
0178          *This.ShowDialog()
0179          // Control returns here when the user clicks the ‘OK’ or ‘Cancel’
0180          // button. Return with the UpdateComplete value.
0181          LeaveSr *This.UpdateComplete
0182      EndFunc
0183
```

Lines 184 - 199: Perform the actual Update operation on the customer number record.

- Line: 189: Read the customer record for update with the customer number provided. Note the record is locked this time.
- Line: 190: If the record was read successfully...
- Line: 191: Populate the record format with values from the user interface.
- Line 192: Peform the Update operation.
- Line 194: If the record was not read successfully...
- Line 195: Put error handling here if the record was not read successfully.
- Line 198: Return the results of the update operation. In this case, true is the hardcoded return value. With rational error handling, the return value would be based on the success of the update operation.

Lines 201 - 225: Perform validation of the values collected with the user interface. Based on error conditions, this routine registers errors with the errProvider ErrorProvider control. This function returns true if the form input validates correctly. The return value of this function is used (in line 69) to condition whether it's OK to continue with the record update process.

The AddUpdate form relies on the ErrorProvider control to display error messages to the user. Errors are registered with the ErrorProvider control with its SetError()

- Line 203: Declare a local integer variable named ErrorCount. The initial value of this variable is zero. As errors are recognized, this value is incremented. After having done all the validation if this value is still zero, no validation errors were encountered.
- Line 206: Calling ClearErrors() removes any previously registered errors from the errProvider ErrorProvider control. This ensures the ValidateForm() routine starts with a clean slate.
- Line 210: If the address (txtCMAddr1.Text) is empty...
- Line 212: Use the ErrorProvider's SetError() method to register an error for the txtCMAddr1 TextBox control. The second argument is the value of the error message.
- Line 213: Increment the ErrorCount variable by 1.
- Line 214: Set the focus to the txtCMAddr1 TextBox control. Controls should be checked for errors in the reverse order in which they are displayed on the screen. That way, the last one focused is the one closest to the top of the tab order.
- Line 216: If the customer name (txtCMName) is empty...
- Line 218: Register an error for the txtCMName TextBox control.
- Line 219: Increment the ErrorCount variable by 1.
- Line 220: Set the focus to the txtCMName TextBox control. Remember, in this routine, the last control to which you set focus wins! Thus, the need to validate controls in reverse tab order.
- Line 224: If the ErrorCount variable value is zero, return true, otherwise return false. The line

```
LeaveSr ( ErrorCount = 0 )
```

is shorthand for

```
If ( ErrorCount = 0 )
    LeaveSr *True
Else
    LeaveSr *False
EndIf
```

In the case of this line:

```
LeaveSr ( ErrorCount = 0 )
```

the value inside the parantheses is evaluated as a Boolean expression and returned to the caller. If ErrorCount is 0, true is returned; otherwise false is returned.

```
0184      BegFunc UpdateRecord Type( *Boolean )
0185          // Update the customer record.
0186          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0187
0188          // Read record for update.
0189          Chain CustByNumber Key( CustomerNumber )
0190          If ( CustByNumber.IsFound )
0191              // If record found, populate its fields and update it.
0192              PopulateRecordFromUI()
0193              Update CustByNumber
0194          Else
0195              // Error handling code here.
0196          EndIf
0197          // Assume the best for now!
0198          LeaveSr *True
0199      EndFunc
0200
0201      BegFunc ValidateForm Type( *Boolean )
0202          // Validate the data on the form.
0203          DclFld ErrorCount       Type( *Integer4 )
0204
0205          // Clear any previously registered errors.
0206          ClearErrors()
0207
0208          // Check for errors in opposite order of data entry
0209          // to ensure the cursor ends up in the top-most error.
0210          If ( txtCMAddr1.Text.Trim() = String.Empty )
0211              // Add an empty address error.
0212              errProvider.SetError( txtCMAddr1, “Address can’t be blank” )
0213              ErrorCount += 1
0214              txtCMAddr1.Focus()
0215          EndIf
0216          If ( txtCMName.Text.Trim() = String.Empty )
0217              // Add an empty name error.
0218              errProvider.SetError( txtCMName, “Name can’t be blank” )
0219              ErrorCount += 1
0220              txtCMName.Focus()
0221          EndIf
0222
0223          // Leave true if ErrorCount = 0, otherwise false.
0224          LeaveSr ( ErrorCount = 0 )
0225      EndFunc
0226
```

Lines 227 - 236 Class constructor. This routine is called implicitly when the AddUpdateForm is instanced. This constructor requires a single parameter of type ASNA.VisualRPG.Runtime.Database. The referenced passed into this constructor is assigned to the AddUpdateForm's local Database object (named pgmDb) on line 235. In this case, the ListForm passes in a reference to its Database object when the AddUpdateForm object is delcared in ListForm.

```
DclFld AddUpdateForm Type( AddUpdate ) New( pgmDB )
```

Passing the Database object into this object is very important. Without doing this, each of the two forms would have their own job on the iSeries. See Chapter 5, The SingleDB Pattern, for more information on this topic.

Lines 237 - 239 Lines generated by the Visual Studio Windows form designer are abridged.

```
0227      BegConstructor Access(*Public)
0228          DclSrParm ParentDB Type( ASNA.VisualRPG.Runtime.Database )
0229          //
0230          // Required for Windows Form Designer support
0231          //
0232          InitializeComponent()
0233
0234          // Assign the DB object from the parent class.
0235          *This.pgmDB = ParentDB
0236      EndConstructor
0237      //
0238      // This abridged listing has omitted the Windows forms-designed generated code.
0239      //
0240  EndClass
```

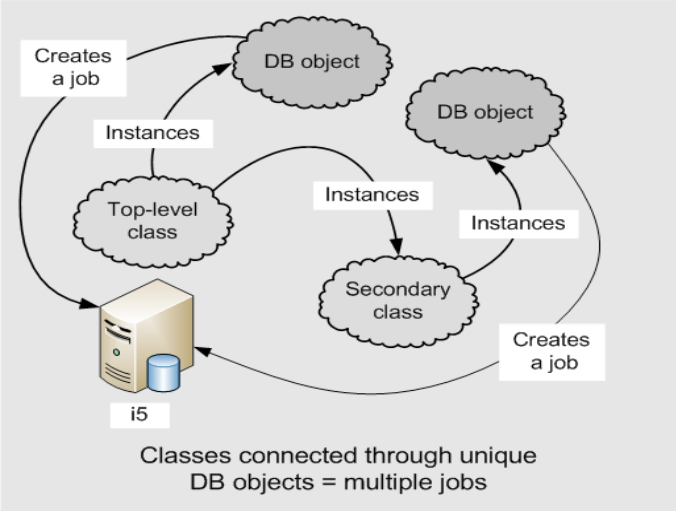
This page is intentionally left blank.

The Singleton DB Pattern

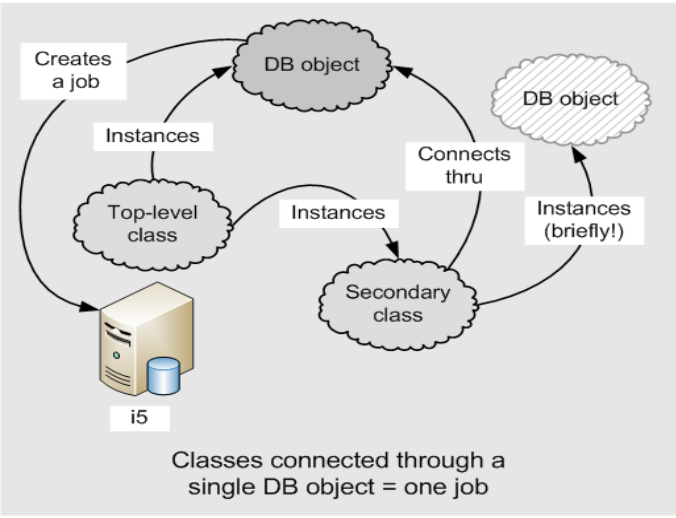
The Singleton DB Pattern reduces the number of connected DB objects for any one program. This pattern improves application performance by sharing a single connected DB object amongst mulitple classes in an application. It hould be used for both Windows and browser-based ASP.NET applications.

The problem:

Consider a class that has a connected DB object (a connected DB object is a DclDB object for which the Connect operation has been issued). This class, let's call it the top-level class (more on this terminology in a moment) instances a secondary class to offload some work. The secondary class also has a connected DB object. With both of these DB objects connected, this program will have two database connections open on the underlying database platform (as shown in the figure to the left). And, this just the tip of the iceberg. It's possible that the problem could grow; should the secondary class instance its own child class with a connected DB object yet another job is created on the database server platform. In short, without using the Singleton DB Pattern, your application will spawn a database server job for *every* database connection operation in your program.



If the DB platform is the i5 this has dramatic performance implications. Creating an i5 job is a relatively time-consuming process. To solve this problem, it's very important to ensure all classes in a project are all connected through the same DB object. It is true that connection pooling can ease the pain of this situation, but not by much. If pooled connections are used to provide multiple database server jobs for a single application, that reduces the number of database connections available to other users—which in turn ultimately does indeed affect performance.



The picture at the left applies both to Windows or ASP.NET applications. In the case of Windows apps, this problem often manifests itself in a multi-form Windows app where each form ends up with its own

database connection. And, because Windows applications would rarely use connection pooling, not only is this process creating unnecessary jobs on the database server, the creation of a new job for each form will impede performance dramatically. In the case of Web applications, the performance impact can be a little sneaky. Because Web apps almost always use connection pooling, you might immediately see much impact with this issue for a low number of users. But as the application scales, you will definitely see unneccessary database server jobs get created, as they are being created, the application will slow down.

The solution

The solution to the problem is to use the Singleton DB Pattern as you design your classes. This pattern simply says that any parent class assigns its DB connection to any classes that the parent instances. While there are several ways to implement the Singleton DB Pattern, perhaps the best way is to have the parent class pass its DB object to children classes in the children classes' constructor. Then, in the children classes, their own copy of the DB object is reassigned to the parent's DB object (we'll see code in a moment).

The picture at the left shows this process at a high level. When the top-level class instances a child class, it passes its DB object by reference to the child. The child has a local DB object and when the child class is instanced there is indeed a copy of this DB object in memory. But, and this is important, the local DB is reassigned to the parent DB object just received in the constructor. After this assignment, the local DB object originally instanced in the secondary class is claimed by .NET's garbage collector.

Having reassigned the DB object in the child class, files will be opened and OS/400 program calls will be performend through the parent's DB object.

Notes on the Singleton DB Pattern

- The parent class should provide a DB object even if it doesn't have any disk files or make any program calls. This DB object is for use assigning the top-level DB to its children.
- The parent class should call a method in all instanced child classes to close files in those children classes. Child classes should always check to see if the DB connection needs to be opened; and open it if necessary.
- Child classes should always check to ensure the assigned DB reference is connected. If it isn't, the child class should connect it. All classes downstream of a top-level parent class should make this test. Given a varied application path, no one class can ever be sure if code before did indeed connect the DB object.
- Child classes should never disconnect a DB connection. That's a job for the parent. A child class never knows if the parent class will need the DB connection when the child class finishes its work.
- Child classes all have a DB object. This object's declared DBName is used at compile-time to locate the disk files specified in the class. At runtime, though, this DB object is reassigned to the reference of the parent DB object. This is usually done in the constructor.
- If a child class itself needs to declare a child class, it simply repeats the pattern, passing a reference to its DB object to the child class. Though its not likely that you'll ever have very many classes instanced at once using a DB object, there is nothing wrong (it fact its desirable) with having them all use the same DB object reference.
- The Singleton DB Pattern works equally well with either Windows or browser-based ASP.NET applications.
- While you could engineer a variation on the Singleton DB Pattern with a Shared(*Yes) DB connection for Windows applications (although we discourage that), under no circumstances should you attempt to use a Shared(*Yes) DB object with ASP.NET applications. This results in one job for the all users of the Web application.
- Part of the reason the Singleton DB Pattern recommends passing the DB object in a class's constructor is that in so doing, you get a compile-time reminder if you forget to pass the DB object reference. Other methods of reassigning the DB object, such as using a DB property in child classes,

This page is intentially left blank.

would easily allow to forget to reassign the child class's DB object.

Code narrative for the Singleton DB Pattern

Line 1 - 18: A top-level parent class. All programs have a top-level parent class. For Windows programs, it's either the starting Windows form code-behind or the class in which a shared Main method lives. For Web applications, it's any one page's code-behind (because only one page exists at a time, different pages in a Web app take turns being the top-level parent class based on what the currently active page is. For class library the top-level parent class is provided by the consuming Windows or Web application. For Web services, the Web service ASMX is the top-level parent class.

Line 2: Declare a DB object in the top-level parent class. This is declared in the top-level parent class even if that class doesn't have any disk files or make program calls. This is the DB object that will be passed on to child classes. Note that the top-level parent class need not connect this DB object—there's no point in connecting a DB object that doesn't need connected. The actual connect can be deferred to a child class. Remember, though. that a child class is really using the DB object owned by the top-level class (after the DB object reassignment in the child class's constructor). When the child class connects the DB object, the DB object being connected is the one owned by the top-level class.

When using the Singleton DB Pattern, children classes are truly using the DB declared in the top-level parent class. That's why no children classes should ever be allowed to disconnect the DB object; they are never aware of following actions that also need a DB connection. That's also why it's important for the top-level parent class to be charged with the responsibility of finally disconnecting the DB object. As a very general rule of thumb, if any one project has more than one occurrence of the Disconnect operation in all of its code, unless multiple database platforms are in use, the Singleton DB Pattern has been violated.

Line 9 - 18: The event handler perfomed before the top-level parent class goes out of scope (before the response leaves the server). This method must call child class-provided methods to close files in those classes and it must disconnect the parent classe's DB object. For Windows apps this method is probably the FormClosing event handler; for Web apps it's probably the Page_Unload event handler. In Web services, this work generally gets done before any one Web method completes.

- Line 11: Call a method in the child class to close its files. It's very import to close files in child classes before those child classes go out of scope.

Line 21 - 45: A secondary child class.

Line 22: The child class's DB object. This object gets reassigned in the child class's constructor.

Lines 25 - 31: A method to see if the DB object needs connected and, if necessary, open files for the child class.

- Line 27: Test to see if the DB object is connected. If not, connected it. Note that although this code is referencing the DB object in this class, by the time this code is executed this DB object is really a reference to the top-level parent class's DB object. Therefore, the connect is effectively connected the DB object in the top-level parent class.
- Line 30: Open any files necessary for the child class here.

Lines 33 - 37: A method to close files in the child class. Because .NET doesn't offer destructors, you must be responsible for closing files before a class goes out of scope. This routine is usually called by the parent class. Don't forget to call it. Otherwise you're likely to get orphan opened files on the database server.

Lines 39 - 44: The child class's constructor.

- Line 40: The constructor receives a parent of type ASNA.VisualRPG.Runtime.DataBase. This is a reference to the DB object in the top-level parent class.
- Line 44: This line assigns the DB reference passed to the constructor to the child class's DB object. See pages 17 and 18 of the Programming with Classes: Concepts and Implementation document for details on assigning an object's reference to another object.

```
0001  BegClass ParentClass Access(*Public)
0002      DclDB pgmDB DBName( “*Public/DG Net Local” )
0003
0004      // Declare an instance of the child class.
0005      DclFLd cc Type( ChildClass ) New( pgmDB )
0006
0007      .. Use the ChildClass as needed.
0008
0009      BegSr Page_Unload Event( *This.Unload )
0010          // Before the parent goes out of scope, be sure to close files in
0011          // the child classes.
0012          cc.CloseData()
0013          // Disconnect DB object.
0014          // Notice that the parent doesn’t have to connect the DB object if
0015          // it doesn’t need to, but the Singleton DB Pattern expects the child to
0016          // check to see if the DB object in play needs to be connected.
0017          Disconnect pgmDB
0018      EndSr
0019  EndClass
0020
0021  BegClass ChildClass Access(*Public)
0022      DclDB pgmDB DBName( “*Public/DG Net Local” )
0023      ... DclDiskFile statements as needed.
0024
0025      BegSr Open Access( *Public )
0026          // Connect the DB object if it isn’t already connected.
0027          If ( NOT pgmDB.IsOpen )
0028              Connect pgmDb
0029          EndIf
0030          ... Open files as needed here.
0031      EndSr
0032
0033      BegSr Close Access( *Public )
0034          // Close all files used by this class.
0035          Close *All
0036          // Child classes _never_ disconnect. That’s a job for the top-level parent.
0037      EndSr
0038
0039      BegConstructor Access(*Public)
0040          DclSrParm pgmDB Type( ASNA.VisualRPG.Runtime.Database )
0041
0042          // Assign the parent’s DB reference to this class’s pgmDB.
0043         *This.pgmDB = pgmDB
0044      EndConstructor
0045  EndClass
```

This page is intentially left blank.

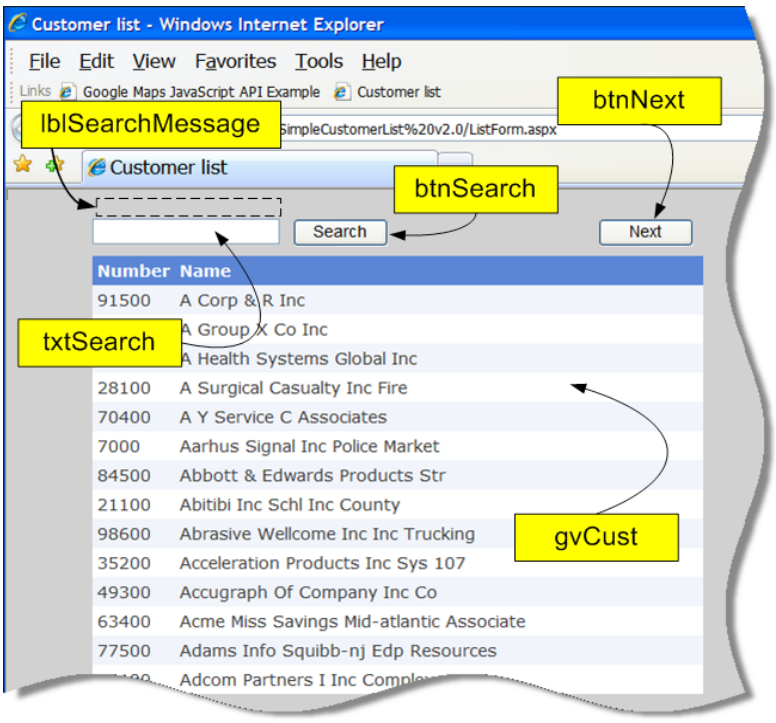
Chapter 6

Simple Web List

The Simple Web List introduces you to how to create a simple subfile-like navigational list in an ASP.NET browser-based program. This exercise introduces several concepts, including:

- The MemoryFile and its Dataset
- Functions
- How to create and program a control-based user interface
- How to program with events, methods, and properties
- How to use program events to control program logics
- How use methods and properties to perform actions and manipulate control behavior
- An introduction into the statelessness of a browser-based application (including understanding IsPostBack and the events that bracket a Web page's lifecycle, Page_Load and Page_Unload).
- How to use the Viewstate to persist values across different instances of the same page.

This document discusses the Web List project and the throught process driving its creation. After discussing briefly how to create the user interface, this document provides a detailed code narrative explaining all aspects of this project's source code.



Program intent: the purpose of this simple project is to introduce several important concepts of creating an AVR for .NET ASP.NET browser-based program. This type of program gets deployed on a Web server and no software installation on the client except for a browser. If you've done the Windows Simple List exercise, you'll be amazed at how similar this ASP.NET code is that Windows code. Like its Windows counterpart, a program like this would generally serve as a navigational front-end for users to select customers for which to perform an action. Although this exercise creates a program targeted specfically for browser use, you'll be amazed at much its code is similiar to its Windows counterpart.

This simple program features two operations: a *Next* operation and a *Search* operation. The Next operation loads the grid with the next group of customers and the Search operation loads the grid with customers starting at a given customer name.

This project uses needs only five controls:

1. A TextBox box for the search value. Name this control txtSearch.
2. A Button for the Search operation. Name this control btnSearch.
3. A Button for the Next operation. Name this control btnNext.
4. A GridView to display customer info in a grid. Name this control gvCust.
5. A Label control to display an error message if the search fails.

Add these controls, assigning their names as you go, so that they are positioned approximately as shown in the figure to the left. (you can make placement and size adjustments later as you go along).

To see the Properties for a given control, select it ("give it focus") and press F4. More detail on setting other properties for these controls follows on the next page.

Setting the control properties

Notes:

Control	Property	Value
txtSearch	Name	txtSearch
btnSearch	Name	btnSearch
lblSearchMessage	Name	lblSearchMessage
	Text	[no value]
btnNext	Name	btnNext
	Text	Next
gvCust	Name	gvCust
	AllowPaging	True
	AutoGenerateColumns	False
	CssClass	FixedSizeText
	Height	[no value]
	PageSize	14
	PagerSettings>Visible	False

Unlike Windows applications, when an ASP.NET Label control doesn't have a value, the name of the control (with brackets around it) is shown as the Label's text.

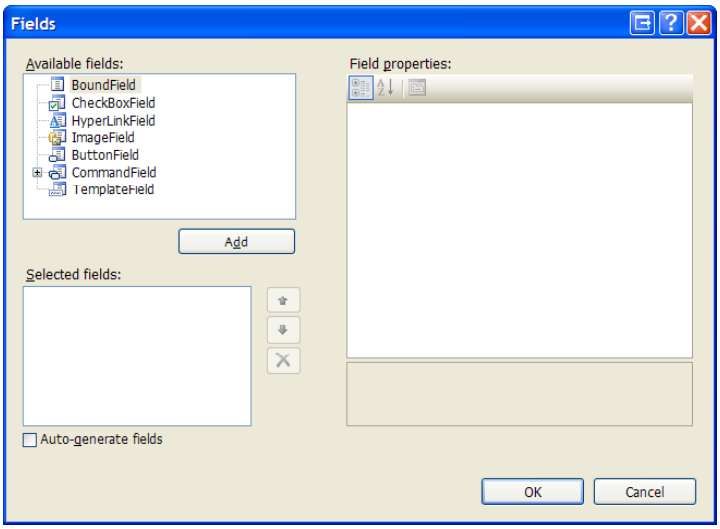
The AutoGenerate property can be changed in two places: it can be changed in the GridView's property window (give the GridView focus and press F4) or it can be changed in the GridView's Columns property dialog (give the GridView focus, press F4, click the Columns property once, and click the elipsis displayed. The AutoGenerateColumns property is available in the lower left-hand corner of this dialog.

In the case of this grid, we set the PageSize to 14 and the PagerSettings>Visible property to false. The PageSize property controls how many rows are in any one grid "page" and the PageSettings>Visible property controls if a small paging footer is displayed. This paging footer is for use with SQL-type databases and the automatic paging of the grid. Rarely do we use automatic paging of the GridView with ASP.NET and AVR for NET because that requires the *entire file* that you are paging to be loaded into memory at once. As an advanced exercise, though, we do have an example that shows how to use automatic paging for those times when you want to page quickly through very small files.

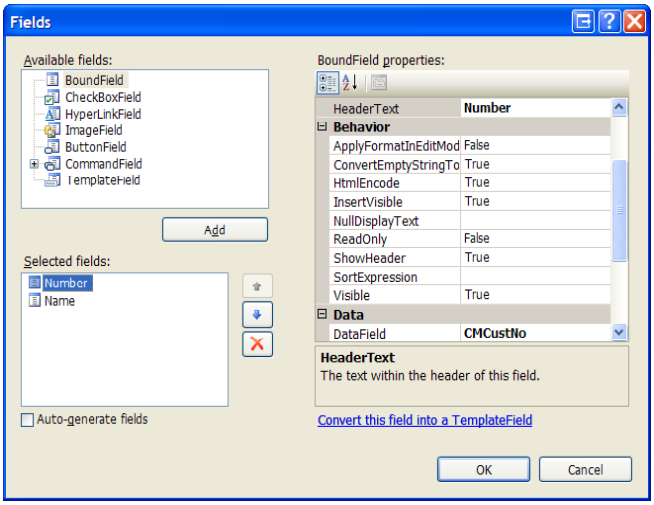
The algorithm used in this program works very well with files of virtually any size. We have customers using this model with file with literally millions of records in them. Try that with automatic paging!

Add two columns to the dvCust GridView

The dvCust GridView needs two columns for this exercise: one for the customer number and one for the customer name. You add these columns from the GridView's Field dialog. To get to this dialog, give the GridView focus, press F4, click the Columns property once, and click the elipsis displayed. The fields dialog is displayed as shown to the left (the image to the left shows the dialog when no fields have been added).



Click the "Add" button and add a BoundField. Set its HeaderText property to "Number" and set its DataField to "CMCustNo" (as shown in the second dialog image to left). In the column's ItemStyle section, set its Width property to "8%." This says to make the number column 8% the total width of the table (remember that the GridView is rendered as HTML, and HTML doesn't naturally deal well with the concept of fixed column widths).



Click the "Add" button a second time and add another BoundField. Set its HeaderText to "Name" and its DataField to "CMName." This column doesn't need a Width set for it, it will expand to fill out the remaining width of the grid. However, for cosmetic purposes, you should probably go to the column's HeaderStyle property section and set its HorizontalAlign property to "Left"to left-justify the column heading.

If you haven't done so already, deselect the Auto-Generate fields checkbox, too.

Column property recap

Control	Property	Value
Number column	HeaderText	Number
	DataField	CMCustNo
	ItemStyle>Width	8%
Name column	HeaderText	Name
	DataField	CMName
	HeaderStyle>HorizontalAlign	Left

This page is intentionally left blank.

Program narrative

With the user interface created, here is a narrative of the source code for the Simple Web List class exercise. This code narrative represents the finished project. It covers all of the concepts in detail that were discussed in class during the creation of this project.

The lines number correspond to the source shown on the facing page and to the source of the downloadable example project. (To ensure that Visual Studio shows line numbers for source code, use the Tools>Text Editor>Basic menu option. In the display section on the right, check "Line Numbers.")

Lines 1 - 7: Default Using statements for a Web program. Namespaces specified by Using statements let you make unqualified references to classes in these namespaces.

Line 13: Declare a DB object. This object is what connects your AVR program to its underlying database server. In AVR, the server platform can either an i5, SQL Server, or the ASNA local DB. In this case, we're using the database specified by the *Public/DG Net Local database name. This is the examples database that installs locally on your PC when you install AVR. Later in this exercise we'll see how to change this so that this program connects to a different database platform (such as the i5 or SQL Server).

Lines 15 - 21: Declare a disk file. The file CMastNewL2, from the Examples library, is opened for indexed, read-only input. Its program name is CustByName, it must be explicitly opened, and uses a field prefix of "Cust_." This field prefix helps provide a pseudo fully-qualified file/field name in the program. See Appendix A for the layout of CMastNewL2's record format. CMastNewL2 is a logical file, keyed first on customer name (CMName) and then customer number (CMCustNo).

Lines 23 - 28: Declare a memory file. A memory file is an in-memory work file. In this case, the memory file has exactly the same layout, including field prefix, as the data file declared above. Records are read from the data file and written to this memory file for the purpose of "binding" records written to the memory file to the DataGridView. Think of this memory file a bit like an externally described, multiple occurrence data structure. Memory files do truly live in memory, thus the more records you write to a memory file, the more memory you are taxing.

Line 31: Declare a global integer variable named RecordsToRead that controls how many records are loaded into the DataGridView at any one time. This value is set dynamically based on the GridView's PageSize property. Note, don't confuse an *Integer4, which is a long integer with a maximum value of 2,147,483,647, with a Zoned 4,0 value. In *Integer4, the 4 represents bytes, not number of digits.

RecordsToRead is global—that is, the variable is scoped to the class rather than to a routine. Thus, it's referenced as **This.RecordsToRead* in the program. **This* indicates the variable is owned by the class, as opposed to being local to a subroutine or function. Prefixing global variables with *This fully qualifies them in your code and helps signal that you're working with a global variable, not a local variable.

Lines 35 - 36: Declare two global variables, one named LastCustomerName and the other named LastCustomerNumber. These variables are declared like the fields Cust_CMName and Cust_CMCustNo in the CMastNewL2's record format These variables will be used by the Next operation to display the next group of customer records.

Lines 41 - 50: Subroutine called when the btnNext button is clicked. Clicking this button puts the next group of records in the dgvCust DataGridView.

- Line 47: Use the SearchForCustomer() function to position the CustByName file at what was the last customer displayed in the grid. This logic carries over the customer on the last row as the customer on the first row of the next display. If you're rather not carry over this last name, use a SetGT here rather than a SetLL in the SearchForCustomer() routine..
- Line 49: Call the LoadGrid() subroutine to put the next group of records in the gvCust GridView.

```
0001  Using System
0002  Using System.Data
0003  Using System.Configuration
0004  Using System.Web
0005  Using System.Web.Security
0006  Using System.Web.UI
0007  Using System.Web.UI.WebControls
0008  Using System.Web.UI.WebControls.WebParts
0009  Using System.Web.UI.HtmlControls
0010
0011  BegClass ListForm Access(*Public) Partial(*Yes) Extends(System.Web.UI.Page)
0012
0013      DclDB pgmDB DBName( "*Public/DG Net Local" )
0014
0015      DclDiskFile  CustByName +
0016            Type( *Input  ) +
0017            Org( *Indexed ) +
0018            Prefix( Cust_ ) +
0019            File( "Examples/CMastNewL2" ) +
0020            DB( pgmDB ) +
0021            ImpOpen( *No )
0022
0023      DclMemoryFile CustMem +
0024            DBDesc( "*Public/DG Net Local" ) +
0025            Prefix( Cust_ ) +
0026            FileDesc( "Examples/CMastNewL2" ) +
0027            ImpOpen( *No ) +
0028            RnmFmt( CustMemR )
0029
0030      // Global variable that controls number of rows in grid.
0031      DclFld RecordsToRead Type( *Integer4 )
0032
0033      // Global variables to store last customer name and number
0034      // current in the grid.
0035      DclFld LastCustomerNumber Like( Cust_CMCustNo )
0036      DclFld LastCustomerName   Like( Cust_CMName )
0037
0038      //---------------------------------------------------------------------------
0039      // Event handlers
0040      //---------------------------------------------------------------------------
0041      BegSr btnNext_Click Access(*Private) Event(*This.btnNext.Click)
0042          // Next button clicked.
0043          DclSrParm sender Type(*Object)
0044          DclSrParm e Type(System.EventArgs)
0045
0046          // Search for customer.
0047          SearchForCustomer( LastCustomerName, LastCustomerNumber )
0048          // Load grid with records.
0049          LoadGrid()
0050      EndSr
0051
```

Note that unlike the Windows version of this program, the Web version doesn't need to explicitly clear the memory file's dataset. Why not? Because each instance of a Web page is a new instance of its underlying code-behind class. Thus (unless you're using some means of caching the results of the memory file across pages) each Web page always starts out with an empty memory file.

Lines 52 - 71 Subroutine called when the Search button is clicked.

Line 58: Call the SearchForCustomer() function passing the customer number entered in the txtSearch TextBox and a zero customer number. If SearchForCustomer() returns true, a customer name matching the search value was found. If the search was successful...

Line 60: Call the LoadGrid() subroutine to put the next group of records in the gvCust GridView.

Line 62: Clear the search name entered.

Line 64: Clear the search message.

Line 65: If the search failed...

Line 67: Set the search message to "Search failed."

Line 69: Position the cursor at the txtSearch TextBox.

Lines 73 - 94: Subroutine called when the server receives the request for the page. This routine get called *every* time a Web page is requested and is effectively the first routine in which code is executed for any one page.

Line 78: Clear the search message.

Line 80: Set RecordsToRead to the value of gvCust GridView's PageSize property. It may seem as if RecordsToRead is a bit unnecessary here—and technically it is. You could just use gvCust.PageSize anywhere in this program where RecordsToRead is used and the program would work just fine. RecordsToRead is included for semantic consistency with the Windows exercise and because when you go through the exercises to separate the file IO and business logical from ListForm's code behind, RecordsToRead has an important role there.

Line 83: Call the OpenData() routine to connect to the database server and open files. Not that in this case, for this program, the DB is connected and files are opened unconditionally. In other programs you might want to defer the connect and opens unti you know they are needed.

Note that lines 78, 80, and 83 are performed uncondtionally—every time a page is loaded.

Line 85 : If this is a new request for this page...

Line 88: Call the LoadGrid() subroutine to put the initial group of records in the gvCust GridView.

Line 89: Otherwise, this page has requested itself again...

Line 92: Restore the values persisted to the Viewstate to the local program variables. More about Viewstate and persisting variables in just a bit.

Lines 96 - 103: Subroutine called just before the response leaves the Web server. Programmers who've done some Windows programming often make the wrong assumption about when this routine gets called. They often assume this routine is like a Windows form's Closing event—that is that it gets called when the browser window is closed. That's not a correct assumption. Remember that browser-based applications are stateless and by the time the user is looking at results in the browser, the server is done processing that response. There is no server-side knowledge of the user closing the browser—that isn't what calls this routine. Rather, these routine is called just prior to the response being sent to the browser. This routine provides a very good place to close files and disconnect the database.

Line 102: Call the CloseData() routine which closes files and disconnects the DB. Let it register that this happens for *every* page! Yes, there is a cost in opening DB connections and files for every page but the stateless environment of the Web demands it. You'll later learn that using connection pooling, especially for the i5, is critical to ASP.NET program performance. Disconnecting a DB connection is the action that returns a DB connection to the connection pool. Opening an i5 pooled connection occurs in a matter of just a few milliseconds; a fresh i5 connection can require up to five or six seconds (on slower i5 hardware). It is very important to disconnect the DB and close files before Page_Unload has finished.

In a related topic, be sure to read the "The Singleton DB Pattern" section. It explains a technique for eliminating unnecessary DB connections.

```
0052      BegSr btnSearch_Click Access(*Private) Event(*This.btnSearch.Click)
0053          // Search button clicked.
0054          DclSrParm sender Type(*Object)
0055          DclSrParm e Type(System.EventArgs)
0056
0057          // If the search was successful...
0058          If ( SearchForCustomer( txtSearch.Text.Trim(), 0 ) )
0059              // Load the grid with records.
0060              LoadGrid()
0061              // Clear search value.
0062              txtSearch.Text = String.Empty
0063              // Clear search message.
0064              lblSearchMessage.Text = String.Empty
0065          Else
0066              // Set search message.
0067              lblSearchMessage.Text = “Search failed.”
0068              // Position cursor at search text.
0069              txtSearch.Focus()
0070          EndIf
0071      EndSr
0072
0073      BegSr Page_Load Access(*Private) Event(*This.Load)
0074          DclSrParm sender Type(*Object)
0075          DclSrParm e Type(System.EventArgs)
0076
0077          // Clear search text.
0078          lblSearchMessage.Text = String.Empty
0079          // Set number of records to read.
0080          RecordsToRead = gvCust.PageSize
0081
0082          // Open DB connection and files.
0083          OpenData()
0084
0085          If ( NOT *This.IsPostBack )
0086              // If page displayed for the first time,
0087              // load grid with initial rows.
0088              LoadGrid()
0089          Else
0090              // Each subsequent time the page is displayed, restore global
0091              // variables from ViewState variables.
0092              RestoreSavedValues()
0093          EndIf
0094      EndSr
0095
0096      BegSr Page_Unload Access(*Private) Event(*This.Unload)
0097          // Raised just before response leaves the server.
0098          DclSrParm sender Type(*Object)
0099          DclSrParm e Type(System.EventArgs)
0100
0101          // Close files and DB connection.
0102          CloseData()
0103      EndSr
0104
```

Lines 108 - 115: CloseData() subroutine that closes the files and disconnects the database server connection. It is very important to close files and disconnect the database server connection before the program ends. Without doing so, you'll files unnecessarily opened and have a potential orphan database job running (especially on the i5). For Web applications, the Page_Unload event handler is a good place to call CloseData().

Lines 117 - 146: LoadGrid() subroutine that read x records from the data file and loads them into the dgvCust DataGridView. To actually get the records into the dgvCust DataGridView, LoadGrid() writes them to the CustMem file. The dgvCust DataGridView ultimately uses the CustMem's DataSet property as its data source.

- Lines 120 -126: A do loop that reads as many records as specified by the global RecordsToRead variable from data file CustByName and writes them to memory file CustMem. Because both files are defined with the same underlying database file, including the field prefix, no manually assignment of field values is necessary here; when a record is read from the data file Cust, the buffer for memory file Cust is implicitly field. If EOF is countered during the read loop, the loop is exited.
- Lines 129 - 130: Save the last customer name and number displayed on the gvCust GridView to global variables LastCustomerName and LastCustomerNumber. These values will be used later for the Next operation. These two lines show how readable field names are when used with a good prefix (in this case, the prefix being "Cust_.")
- Lines 134 - 135: In ASP.NET applications, variables do not naturally persist across page views. Thus, to persist a variable's value, you must explicitly persist it somewhere. You'll later learn that there are actually several ways to persist variables. However, one of the easiest ways is to save them to the page's Viewstate. The Viewstate is simply a collection of value pairs (with a key and a value) that are ultimately saved with the content of the page in hidden field in the ASPX page. These two lines of code save the LastCustomerNumber and the LastCustomerName variables to the Viewstate. Note that adding variables to the Viewstate requires no declaration or other setup you just assign a key value a value. Note also that you don't even tell the Viewstate what the data types are of the values you're putting in it. You'll soon see that, as backwards as this seems, you must specify their types when you fetch variable values back out of the Viewstate.
  It's important to note that Viewstate variables are only saved for the page on which they are declared. They *do not s*pan across other pages. Typically, for that you would use Session variables. You'll use those in the List/Update exercise.
- Line 138: Assign the gvCust GridView's DataSource property.
- Line 142: Unlike the Windows DataGridView, the ASP.NET GridView requires a second step to display data in the grid; you must call the grid's DataBind() method after assigning its datasource. This is actually the step that causes the grid to render itself as an HTML table in your page's response. Omit the DataBind() step and you'll get an empty grid!
- Line 145: Use the results of the MoreRecordsForward() function to toggle btnNext's Enabled property. More on this in a bit.

```
0105      //------------------------------------------------------------------------
0106      // Private subroutines and functions.
0107      //------------------------------------------------------------------------
0108      BegSr CloseData
0109          // Close CustByName file.
0110          Close CustByName
0111          // Close CustMem file.
0112          Close CustMem
0113          // Disconnect DB connection.
0114          Disconnect pgmDB
0115      EndSr
0116
0117      BegSr LoadGrid
0118          // Starting at current file position, read next x records,
0119          // writing them to the memory file then binding them to the grid.
0120          Do FromVal( 1 ) ToVal( *This.RecordsToRead )
0121              Read CustByName
0122              If ( CustByName.IsEof )
0123                  Leave
0124              EndIf
0125              Write CustMem
0126          EndDo
0127
0128          // Save last customer name and number displayed.
0129          LastCustomerName   = Cust_CMName
0130          LastCustomerNumber = Cust_CMCustNo
0131
0132          // Save last customer name and number displayed to
0133          // ViewState to persist values across page instances.
0134          ViewState[ “lastcustomername”   ] = Cust_CMName
0135          ViewState[ “lastcustomernumber” ] = Cust_CMCustNo
0136
0137          // Set the datasource for the DataGridView.
0138          gvCust.DataSource = CustMem.GetFileData()
0139          // “Bind” the datasource data to the grid.
0140          // This step causes the grid’s HTML to be
0141          // generated (as an HTML table).
0142          gvCust.DataBind()
0143
0144          // Should the Next button be enabled?
0145          btnNext.Enabled = MoreRecordsForward()
0146      EndSr
```

Lines 148 - 154: MoreRecordsForward() function that returns true if there are more records to read foward from the last customer currently showing in the grid; otherwise this function returns false.

Lines 156 - 163: OpenData() subroutine that connects to the database server and opens files. In the real world, you'll want a little more error handling than is shown here (a little more than the *none* shown here!). We'll discuss robust error handling later.

Lines 165 - 169: Subroutine called to restore the variable values that were previously saved to the Viewstate. This routine is called in the NOT *This.IsPostBack portion of the Form_Load routine (at line 92). Thus, saved values are restored so that, for all intents and purposes, your code behaves as though they never went away (they did, but these two lines brought them back!).

Note also that these values must be cast to a string to restore them. You don't specify a value's type when you put into the Viewstate, but you must specify it's type (this process is called casting) as you extract if from Viewstate. In this case, it makes perfect sense that the customer name be cast as a string—it is a string! But what at the customer number? It's numeric, why is it cast as a string? The short story is that AVR's = assignment operator is, like RPG's MOVE operation, part assignment operator and part conversion operator. See the Simple variable assignments section near the bottom of page 8 of the "Introducing AVR for .NET" document for more details on this.

Lines 171 - 181: Function called to determine if the CustByName file can be positioned at the given customer name and number. If the SetLL with these two value succeeds, true is returned; otherwise false is returned.

```
0147
0148      BegFunc MoreRecordsForward Type( *Boolean ) Access( *Public )
0149          // See if there are records beyond the last row.
0150
0151          SetGT CustByName Key( *This.LastCustomerName, *This.LastCustomerNumber )
0152          LeaveSr CustByName.IsFound
0153      EndFunc
0154
0155      BegSr OpenData Access( *Public )
0156          // Open DB connection.
0157          Connect pgmDB
0158          // Open CustByName file.
0159          Open CustByName
0160          // Open MemoryFile.
0161          Open CustMem
0162      EndSr
0163
0164      BegSr RestoreSavedValues
0165          // Restore global variables from ViewState.
0166          LastCustomerName   = ViewState[ “lastcustomername”   ].ToString()
0167          LastCustomerNumber = ViewState[ “lastcustomernumber” ].ToString()
0168      EndSr
0169
0170      BegFunc SearchForCustomer Type( *Boolean )
0171          // Search for a customer.
0172          DclSrParm CustomerName   Like( Cust_CMName )
0173          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0174
0175          // Set lower limits at name and number provided.
0176          SetLL CustByName Key( CustomerName, CustomerNumber )
0177
0178          // Return value is boolean indicating if search was successful.
0179          LeaveSr CustByName.IsFound
0180      EndFunc
0181
0182  EndClass
```

This page is intentionally left blank.

Chapter 7

Simple Web list with update panel

The AddUpdate form lets the user change values for a given customer. It is displayed when a user double-clicks a customer name in a row in the customer list from the previous exercise.

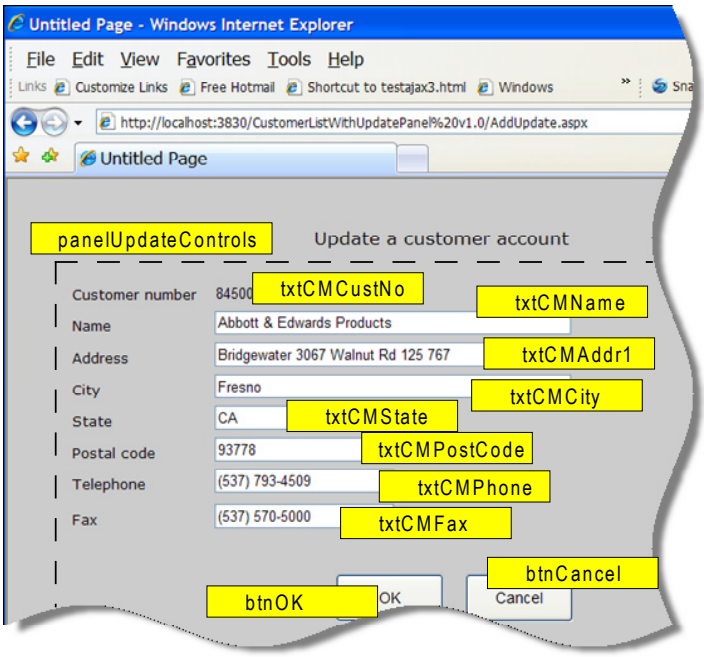
Program intent: the purpose of this simple project is to introduce some of the important concepts of navigating from one from Web form to another and using AVR file IO operations to write changes to disk.

This part of the program illustrates how to:

- Do a database update with AVR
- Open a form from a another form, passing information back and forth
- Perform input validation
- Lay the groundwork for optimistic record locking. This version of the program doesn't fully implement optimistic record locking, but it doesn't lock the record until it's ready for update. Another example will show you the full details on optimistic locking.

To build this program, copy the Web version of the SimpleCustomerList project to another folder. Add a new Web form named AddUpdate.aspx. The AddUpdate Web form in this project needs these controls:

1. A Panel on which all of the other controls are placed. Name this panel panelUpdateControls. More on this panel in a moment.
2. A TextBox for the customer number. Name this control txtCMCustNo
3. A TextBox for the customer name. Name this control txtCMName.
4. A TextBox for the customer address Name this control txtCMAddr1.
5. A TextBox for the customer city. Name this control txtCMCity.
6. A TextBox for the customer state. Name this control txtCMState.
7. A TextBox for the customer postal code. Name this control txtCMPostCode.
8. A TextBox for the customer phone. Name this control txtCMPhone.
9. A TextBox for the customer fax. Name this control txtCMFax.
10. An OK button. Name this control btnOK.
11. A cancel button. Name this control btnCancel.



Also, add a Label control to the left of each of the eight TextBoxes and add a Label control near the top of the form for the form heading. Use the default name for these controls, but assign their Text property as shown below. Also not shown on the form to the left, but needed, is a RequiredFieldValidator named validateCMName and a CustomValidator named validateCMCity. Put these two validators near the right-hand edge of the txtCMname and txtCMcity TextBoxes. Also add a ValidationSummary control in the lower-left hand corner of the form. It doesn't have to have any properties set.

Setting the control properties

In addition to the control names (and the labels' Text properties) set these controls' properties as show:

Place the panelUpdateControls Panel on the Web form first. Then place all the other controls on it. Don't worry too much about the panel's size, it can be changed later, if needed. It should be large enough to hold all of the controls. This panel is needed to allow the Enter key to act as a click on the OK when it is pressed. You'll see how to hook this up in a moment.

Control	Property	Value
panelUpdateControls	ID	panelUpdateControls
txtCMName	ID	txtCMName
txtCMAddr1	ID	txtCMAddr1
txtCMCity	ID	txtCMCity
txtCMState	ID	txtCMState
txtCMPostCode	ID	txtCMPostCode
txtCMPhone	ID	txtCMPhone
txtCMFax	ID	txtCMFax
btnOK	ID	btnOK
	Text	OK
btnCancel	ID	btnCancel
	Text	Cancel
validateCMName	id	validateCMName
	ControlToValidate	txtCMName
	ErrorMessage	Please enter a customer name
	Text	*
validateCMCity	id	validateCMCity
	ErrorMessage	City cannot have numbers in it
	Text	*

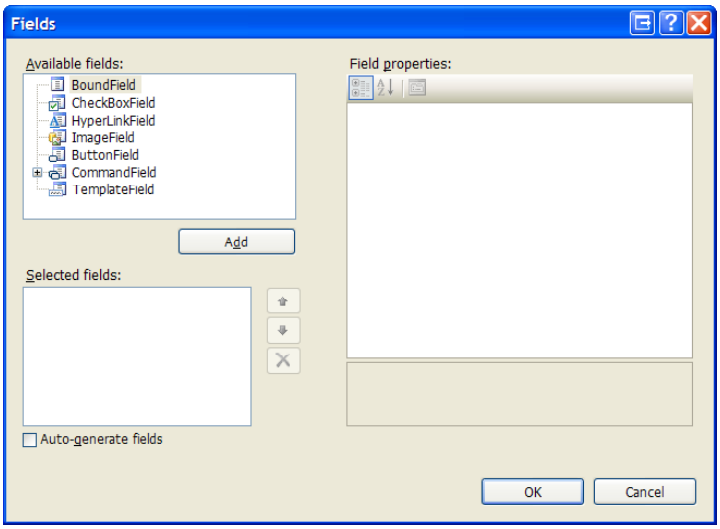
The two validator controls are used to show how to perform data validation. The ValidatorSummary control is a way to show a recap of the errors on the form. If you leave the Text property blank on the two validators, the ErrorMessage text shows next to the TextBox (and in the summary list). If you put a value in the Text property, that property shows next to the TextBox and the summary list shows the ErrorMessage property. You can try out these difference combinations to see how you'd like to display your validation errors.

By default, the RequiredFieldValidator performs client-validation (that is, it does its checking with JavaScript). However, interesting enough, if you set the RequiredValidator's EnableClientScript property to false, that validator also "injects" the server-side code necessary to perform the required field validation. You don't write any code for the RequiredFieldValidator (or its cousins, the RangeValidator, the RegularExpressionValidator, the CompareValidator).

The CustomerValidator, on the other hand, performs your customer validation code. It is While you can hook up JavaScript to this controls, they are most frequently used with server-side AVR code (as is this example).

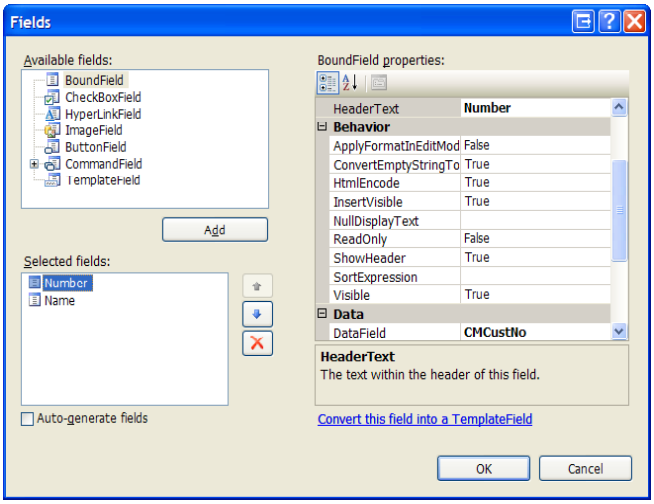
Add two columns to the dvCust GridView

The dvCust GridView needs two columns for this exercise: one for the customer number and one for the customer name. You add these columns from the GridView's Field dialog. To get to this dialog, give the GridView focus, press F4, click the Columns property once, and click the elipsis displayed. The fields dialog is displayed as shown to the left (the image to the left shows the dialog when no fields have been added). If you're building a project that starts with the Simple Web List, delete the Name column (select it and click teh red X) and jump ahead to the "Add a csutomer name column" section.



Add a customer number column

Click the "Add" button and add a BoundField. Set its HeaderText property to "Number" and set its DataField to "CMCustNo" (as shown in the second dialog image to left). In the column's ItemStyle section, set its Width property to "8%." This says to make the number column 8% the total width of the table (remember that the GridView is rendered as HTML, and HTML doesn't naturally deal well with the concept of fixed column widths).



Add a customer name column

Click the "Add" button and add a ButtonField. Set its HeaderText to "Name" and its DataField to "CMName." Set the CommandName property to "SELECT". This is very important. If you forget to set the CommandName property, when a name is clicked the row is highlighted but that's all that happens! This column doesn't need a Width set for it, it will expand to fill out the remaining width of the grid. For cosmetic purposes, you should also probably go to the column's HeaderStyle property section and set its HorizontalAlign property to "Left"to left-justify the column heading.

If you haven't done so already, deselect the Auto-Generate fields checkbox, too.

Column property

Control	Property	Value
Number column	HeaderText	Number
	DataField	CMCustNo
	ItemStyle>Width	8%
Name column	HeaderText	Name
	DataField	CMName
	HeaderStyle>HorizontalAlign	Left
	CommandName	SELECT

Program narrative for changes to ListForm.aspx.vr

The ListForm.aspx user interface doesn't change in this example. However, you do need to add code to its ListForm.aspx.vr codebehind class in four primary places. These additions shown on the facing pages. The line numbers in the code to the right are for referencing the lines of code to change or add—they *do not* show where to add these lines in ListForm.aspx.vr. The narrative below details these additions.

Lines 1- 25: Subroutine called when a customer name is clicked on the GridView. This routine will determine what customer number was selected for update, as well as track the current customer name and number at the top of the list (for cancel operation purposes). These values will are stored in Session variables (more about these later in this document) and then control is redirected to the AddUpdate.aspx page.

- Line 7: Declare an integer variable in which to store the row number of the selected row (the row the user clicked)
- Line 10: Set the SelectedRow value from the GridView's SelectedIndex property.
- Line 14 - 15: Assign the "cmcustno" value from the SelectedRow in the DataKeys structure to the "customernumber" session variable. This value will be used by the Add/Update form to know which customer is being updated.
- Line 17 - 18: Assign the "cmcustno" value from the zeroth row in the DataKeys structure to the "cancelcustomernumber" session variable.
- Line 20 - 21: Assign the "cmname" value from the zeroth row in the DataKeys structure to the "cancelcustomername" session variable. This value and the previous value assigned in lines 17 - 18 are used to reposition the customer list as it was if the update operation is canceled.

Notes on the DataKey type

The DataKey type is a special data structure (provided by the GridView) that automatically stores specified key values as data is bound to the GridView. These key values are, as you'll see easy to extract. DataKey is a multiple dimensional array with an element for each row. In this case, two key values are saved for each row. These key values are are named "cmcustno" and "cmname" (you'll see how they get those names when we discuss lines 67-75). The figure below shows an abbreviated image of what this structue might look like for this first three rows of the grid. Data is added to the DataKey structure implicitly as data is bound to the GridView. The cmcustno and cmname values are retrieved in lines 14 - 20.

Row	cmcustno	cmname
0	5600	Bob Jones
1	2100	William Smith
2	45	Julia Andover

Lines 40 - 43: Add two subroutine calls in Page_Load. The AssignDataKeyNames (Line 41) assigns key field names to the GridView's DataKeyNames property. This property tells the GridView what key values to store in the DataKeys structure. The SetListStartPosition() routine attempts to position the customer file at a specified customer name and number (as indicated by session variables "cancelcustomername" and "cancelcustomernumber").

Note that the rest of Page_Load is unchanged from ListForm.aspx.vr.

```
0001      BegSr gvCust_SelectedIndexChanged Access(*Private) +
0002                                        Event(*This.gvCust.SelectedIndexChanged)
0003          DclSrParm sender Type(*Object)
0004          DclSrParm e Type(System.EventArgs)
0005
0006          // Declare a variable for the selected row position.
0007          DclFld SelectedRow Type( *Integer4 )
0008
0009          // Get the selected row's ordinal position.
0010          SelectedRow = gvCust.SelectedIndex
0011
0012          // Fetch values from the given row and column for
0013          // key values to pass as session variables to the next page.
0014          Session[ "customernumber" ] = +
0015                gvCust.DataKeys[ SelectedRow ][ "cmcustno" ].ToString()
0016
0017          Session[ "cancelcustomernumber" ] = +
0018                gvCust.DataKeys[ 0 ][ "cmcustno" ].ToString()
0019
0020          Session[ "cancelcustomername" ] = +
0021               gvCust.DataKeys[ 0 ][ "cmname"   ].ToString()
0022
0023          // Redirect control to the AddUpdate.aspx page.
0024          Response.Redirect( "AddUpdate.aspx" )
0025      EndSr
0026
0027      BegSr Page_Load Access(*Private) Event(*This.Load)
0028          DclSrParm sender Type(*Object)
0029          DclSrParm e Type(System.EventArgs)
0030
0031          // Clear search text.
0032          lblSearchMessage.Text = String.Empty
0033          // Set number of records to read.
0034          RecordsToRead = gvCust.PageSize
0035
0036          // Open DB connection and files.
0037          OpenData()
0038
0039          If ( NOT *This.IsPostBack )
0040              // Assign key values to save for each grid row.
0041              AssignDataKeyNames()
0042              // Attempt to set the initial position of the list.
0043              SetListStartPosition()
0044              // If page displayed for the first time,
0045              // load grid with initial rows.
0046              LoadGrid()
0047          Else
0048              // Each subsequent time the page is displayed, restore global
0049              // variables from ViewState variables.
0050              RestoreSavedValues()
0051          EndIf
0052      EndSr
0053
```

Program narrative for changes to ListForm.aspx.vr (continued)

Lines 54- 62: Assign an array of field names to the GridView's DataKeyName's property.

- Line 57: Declare a string array named KeyNames with two elements.
- Line 58: Assign "cmcustno" to the first element.
- Line 59: Assign "cmname" to the second element.
- Line 61: Assign the array to the GridView's DataKeyNames property. This array property defines what values from the underlying datasource are automatically stored in the DataKeys structure of the GridView (as data is bound to the GridView).

Lines 64 - 78: Attempt to set the starting position of the customer list in the GridView. If session variables "customernumber" and "customername" have values, these values are used to position the customer file before populating the GridView.

- Lines 66 - 67: Check to see if the session variables "customernumber" and "customername" exist.
- Lines 69 - 70: If those session variables do exist, assign the Cust_CMName and Cust_ CMCustNo values from those session variables. Note the use of ToString() to cast the values as strings during the assignment.
- Lines 71 - 72: If the SearchForCustomer() funciton returned true, the specified customer name and number were found. Therefore, position the file at that customer name and number.
- Lines 75 - 76: Remove the "customernumber" and "customername" session variables.

```
0054      BegSr AssignDataKeyNames
0055          // Specify field names to save to later fetch key values for a given
0056          // GridView row.
0057          DclArray KeyNames Type( *String ) Dim( 2 )
0058          KeyNames[ 0 ] = “cmcustno”
0059          KeyNames[ 1 ] = “cmname”
0060
0061          gvCust.DataKeyNames = KeyNames
0062      EndSr
0063
0064      BegSr SetListStartPosition
0065          // Attempt to set the list starting position.
0066          If ( Session[ “customernumber”  ] <> *Nothing ) AND +
0067             ( Session[ “customername”    ] <> *Nothing )
0068
0069              Cust_CMName   = Session[ “customernumber” ].ToString()
0070              Cust_CMCustNo = Session[ “customername”   ].ToString()
0071              If ( NOT SearchForCustomer( Cust_CMName, Cust_CMCustNo ) )
0072                  SetLL CustByName Key( *Start )
0073              EndIf
0074
0075              Session.Remove( “customernumber” )
0076              Session.Remove( “customername” )
0077          EndIf
0078      EndSr
```

Program narrative for AddUpdate.aspx.vr

Lines 1 - 10: Default Using statements for a Web program. Namespaces specified by Using statements let you make unqualified references to classes in these namespaces.

Line 11: A Using statement for the System.Text.RegularExpressions namespace. Its RegEx.Replace() and RegEx.Match() methods are used for validating and formatting purposes in this program.

Line 13: The code-behind class begins.

Lines 15 - 24: Declare a DB connection and the CMastNewL1 diskfile. This file is indexed on customer number.

Line 27: A global variable that stores the customer number being updated.

Lines 31 -32: Global variables to store the customer name and number for this form's Cancel operation. When the Cancel but is clicked, these two values are passed back to the selection list to cause it to reposition itself back to where it was when the user requested the Update operation.

Note that these three global variables are stored in the Viewstate to persist their values across page instances. See the SaveGlobalVariables() and RestoreGlobalVariables() routines.

Lines 37 - 51: Subroutine called when the Cancel button is clicked.

- Lines 43 - 44: The customer name and number saved for cancel operation are assigned to two session variables. These two session variables are used by the customer list form to reposition it to its original order when the user requested the update operation.
- Line 47: Call the RemoveSessionVariables() routine to remove the session variables that this page needed.
- Line 50: Redirect to the ListForm.aspx page.

```
0001  Using System
0002  Using System.Data
0003  Using System.Configuration
0004  Using System.Collections
0005  Using System.Web
0006  Using System.Web.Security
0007  Using System.Web.UI
0008  Using System.Web.UI.WebControls
0009  Using System.Web.UI.WebControls.WebParts
0010  Using System.Web.UI.HtmlControls
0011  Using System.Text.RegularExpressions
0012
0013  BegClass AddUpdate Partial(*Yes) Access(*Public) Extends(System.Web.UI.Page)
0014
0015      DclDB pgmDB DBName( "*Public/DG Net Local" )
0016
0017      DclDiskFile  CustByNumber +
0018            Type( *Update ) +
0019            Org( *Indexed ) +
0020            Prefix( Cust_ ) +
0021            File( "Examples/CMastNewL1" ) +
0022            DB( pgmDB ) +
0023            ImpOpen( *No ) +
0024            AddRec( *Yes )
0025
0026      // The customer number being updated.
0027      DclFld CustomerNumber       Like( Cust_CMCustNo )
0028
0029      // The name and number of the customer to which the grid is positioned
0030      // if the update is canceled.
0031      DclFld CancelCustomerName   Like( Cust_CMName )
0032      DclFld CancelCustomerNumber Like( Cust_CMCustNo )
0033
0034      //------------------------------------------------------------------------
0035      // Event handlers.
0036      //------------------------------------------------------------------------
0037      BegSr btnCancel_Click Access(*Private) Event(*This.btnCancel.Click)
0038          // Fires when cancel button clicked.
0039          DclSrParm sender Type(*Object)
0040          DclSrParm e Type(System.EventArgs)
0041
0042          // Set customer name and number for grid positioning.
0043          Session[ "CustomerName"   ] = *This.CancelCustomerName
0044          Session[ "CustomerNumber" ] = *This.CancelCustomerNumber
0045
0046          // Remove unnecessary session variables.
0047          RemoveSessionVariables()
0048
0049          // Display the Default.aspx page.
0050          Response.Redirect( "ListForm.aspx" )
0051      EndSr
0052
```

Lines 53 - 74: Subroutine called when the OK button is clicked.

Line 59:	Call the ValidateForm() function. This function performs all server-side custom validation. If it finds any errors, it sets the appropriate validator control's IsValid property to false—which in turn sets the global Page.IsValid property to false.
Line 60:	If the Page.IsValid property is true, then continue with the update process. Page.IsValid is set implicitly by any of the validators controls.
Line 62:	
Lines 65 - 66:	
Line 69:	
Line 72:	

Lines 76 - 98: Subroutine called when the server receives the request for the page. This routine get called *every* time a Web page is requested and is effectively the first routine in which code is executed for any one page.

Line 83:	Set the DefaultButton property for the panelUpdateControls control. This could be done through the property window, but the value is case-sensitive—because it's ultimately associated with some JavaScript that gets generated for the page. Most of the ASP.NET controls have the ClientID property which resolves to the final HTML ID of the control.
Line 85:	If this is a new request for this page...
Line 88 - 90:	Populate the global class variables CustomerNumber, CancelCustomerNumber, and CancelCustomerName with values from Session variables. These session variables were populated in the customer list program. These three values comprise the inputs to this Web page from the previous Web page. Session variables are used in a similar fashion to Viewstate variables, however unlike Viewstate variables, Session variables are stored in memory on the Web server (by default, ASP.NET does include other ways to reconfigure where and how session variable contents are stored—but most of the time they are stored in memory on the server). Session variables are unique to any one browser instance. Thus, if the same user opens two browser instances, Session variables ensure each instance stores its own unique values. Session variables will be covered in more detail later.
Line 93:	Call the function to read a customer record. This function reads the customer as indentified by the customer number passed as the single argument to it. More on this function in a bit.
Line 94:	Otherwise this page has requested itself again...
Line 96:	Restore the values persisted to the Viewstate to the local program variables.

Lines 100 - 111: Subroutine called just before the page response is rendered to the client. This event handler immediately precedes the Form_Unload subroutine. The difference is that the Viewstate is not available in the
Form_Unload routine but it is available in the Page_PreRender routine.

Line 110:	Call the SaveGlobalVariables() routine to save class variables to the Viewstate.

```
0053      BegSr btnOK_Click Access(*Private) Event(*This.btnOK.Click)
0054          // Fires when OK button clicked.
0055          DclSrParm sender Type(*Object)
0056          DclSrParm e Type(System.EventArgs)
0057
0058          // Perform server-side form validation.
0059          ValidateForm()
0060          If ( Page.IsValid )
0061              // Update record with new data.
0062              UpdateRecord( CustomerNumber )
0063
0064              // Set customer name and number for grid positioning.
0065              Session[ "customername"   ] = Cust_CMName
0066              Session[ "customernumber" ] = Cust_CMCustNo
0067
0068              // Remove unnecessary session variables.
0069              RemoveSessionVariables()
0070
0071              // Display the Default.aspx page.
0072              Response.Redirect( "ListForm.aspx" )
0073          EndIf
0074      EndSr
0075
0076      BegSr Page_Load Access(*Private) Event(*This.Load)
0077          // Fires each time page loads.
0078          DclSrParm sender Type(*Object)
0079          DclSrParm e Type(System.EventArgs)
0080
0081          // Set button "clicked" when Enter is pressed when the panelUpdateControls
0082          // has focus.
0083          panelUpdateControls.DefaultButton = btnOK.ClientID
0084
0085          If ( NOT Page.IsPostBack )
0086              // Occurs the first time the page loads.
0087              // Load program global variables from session variables.
0088              *This.CustomerNumber       = Session[ "customernumber"       ].ToString()
0089              *This.CancelCustomerNumber = Session[ "cancelcustomernumber" ].ToString()
0090              *This.CancelCustomerName   = Session[ "cancelcustomername"   ].ToString()
0091
0092              // Read a customer record for display.
0093              ReadRecord( *This.CustomerNumber )
0094          Else
0095              // Restore global variables values.
0096              RestoreGlobalVariables()
0097          EndIf
0098      EndSr
0099
0100      BegSr Page_PreRender Access(*Private) Event(*This.PreRender)
0101          // Fires just before server finishes with page response.
0102          DclSrParm sender Type(*Object)
0103          DclSrParm e Type(System.EventArgs)
0104
0105          // PreRender is the last chance to reference Viewstate variables.
0106          // By the time PageUnload fires the Viewstate is no longer available
0107          // to server-side code.
0108
0109          // Save global variables to persist across page instances.
0110          SaveGlobalVariables()
0111      EndSr
0112
```

Lines 116 - 122: CloseData() subroutine that closes the files and disconnects the database server connection. It is very important to close files and disconnect the database server connection before the program ends. Without doing so, you'll files unnecessarily opened and have a potential orphan database job running (especially on the i5).

Lines 124 - 129: OpenData() subroutine that connects to the database server and opens files. In the real world, you'll want a little more error handling than is shown here (a little more than the *none* shown here!). We'll discuss robust error handling later.

Lines 131 - 140: PopulateRecordFromUI() subroutine that populates the record format fields with values from the user interface controls on the Web form. Most of these value are string data types, so nothing but a simple assignment is needed. However, in the case of the Cust_CMFax value, it is a packed 10,0 value. LIke all other TextBoxes, the txtCMFax TextBox's Text property is a string value. It's likely that the user would enter a fax number with formatting characters (for example, the user might enter the value like this:
(560) 905-9087). Thus, the non-numeric characters need to be removed from the Text property before assigning the value to the packed value. It does this by calling the RemoveNonNumericCharacters() function. We'll discuss that function later in this document.

Lines 142 - 152: PopulateUIFromRecord() subroutine that populates the user interface controls with values from the most recently read CustByNumber record format. When assigning character values to control properties, you'll generally want to Trim() them (remove trailing blanks). If you don't, the trailing blanks will be a part of the control value—and this is often off-putting to a user (for example, she'll need to backspace through the blanks to change the last character of a field).

For other data types, formatting may be necessary. Line 131 shows that the custom numeric formatting capabilities of the ToString() are used to format the fax number from its intrinsic 10,0 numeric value to a more pleasing representation of a fax number. If you'd rather not use ToString() to format a value, you can also use RPG edit codes and edit words. For example, you could use this:

```
txtCMFax.Text   = %EDITW( Cust_CMFax, "0(   )&    -    " )
```

to format the fax number appropriately. Having said that, once you get familar with ToString()'s formatting capabilities, you'll find that an intuitive, easy way to format values. You can read more detail on ToString()'s custom numeric formatting capabilities as this link:

http://msdn.microsoft.com/library/?url=/library/en-us/cpguide/html/cpconcustomnumericformatstrings.asp

While we're on the subject of formatting values, if you need to format a date data type, ToString() offers powerful date formatting as well. Read about it at the link below:

http://msdn.microsoft.com/library/?url=/library/en-us/cpguide/html/cpconcustomdatetimeformatstrings.asp

```
0113      //-------------------------------------------------------------------------
0114      // Subroutines and functions.
0115      //-------------------------------------------------------------------------
0116      BegSr CloseData Access( *Public )
0117          // Close CustByNumber.
0118          Close CustByNumber
0119
0120          // Disconnect DB connection.
0121          Disconnect pgmDB
0122      EndSr
0123
0124      BegSr OpenData Access( *Public )
0125          // Open DB connection.
0126          Connect pgmDB
0127          // Open Cust file.
0128          Open CustByNumber
0129      EndSr
0130
0131      BegSr PopulateRecordFromUI
0132          // Populate the record format from the user interface.
0133          Cust_CMName     = txtCMName.Text
0134          Cust_CMAddr1    = txtCMAddr1.Text
0135          Cust_CMCity     = txtCMCity.Text
0136          Cust_CMState    = txtCMState.Text
0137          Cust_CMPostCode = txtCMPostCode.Text
0138          Cust_CMFax      = RemoveNonNumericCharacters( txtCMFax.Text )
0139          Cust_CMPhone    = txtCMPhone.Text
0140      EndSr
0141
0142      BegSr PopulateUIFromRecord
0143          // Populate the user interface from the record format.
0144          txtCMCustNo.Text   = CustomerNumber.ToString()
0145          txtCMName.Text     = Cust_CMName.Trim()
0146          txtCMAddr1.Text    = Cust_CMAddr1.Trim()
0147          txtCMCity.Text     = Cust_CMCity.Trim()
0148          txtCMState.Text    = Cust_CMState.Trim()
0149          txtCMPostCode.Text = Cust_CMPostCode.Trim()
0150          txtCMFax.Text      = Cust_CMFax.ToString( “(000) 000-0000” )
0151          txtCMPhone.Text    = Cust_CMPhone.Trim()
0152      EndSr
0153
```

Lines 154 - 175: A function to read a record. Note that this function is very specifically for reading the record to display its values to the user (done with PopulateUIFromRecord() after the record is read). This routine *does not* lock the record. When its time to update the record, the record will be read again for update purposes. A later exercise discusses how to manage optimistic record locking.

Note that ReadRecord() is a function. In this exercise it unconditionally returns true. Later, when error handling is hooked up, it won't be hardwired to return true.

Lines 177 - 185: A function to remove non-numeric characters from a string. This function uses a regular expression to strip non-numeric characters from a given string. Regular expressions offer a succinct way to perform pattern matching and character replacement on string values. In this case, the line:

```
LeaveSr Regex.Replace( Value, "[^0-9]", String.Empty )
```

uses the Replace() method to search Value for any character matching the regular expression [^0-9] and replaces any found with String.Empty (a .NET intrinsic value for ""). [^0-9] is the regular expression that matches any occurrence of a non-numeric character.

If this class didn't have a

```
Using System.Text.RegularExpressions
```

at the top of its code, we couldn't have made the shorthand call to RegEx.Replace(), we would have had to make a fully qualified call like this:

```
LeaveSr System.Text.RegularExpressions.Regex.Replace( Value, "[^0-9]", String.Empty )
```

As you can probably imagine, there is a lot to regular expressions. They will be covered in more depth later.

Lines 187 - 190: When you put values in Session variables its important to remove them as soon as possible (the default, and most widely-used, ASP.NET Session model is to save session variables to memory on the Web server). This routine removes the two session variables that aren't needed outside of these page.

Lines 193 - 198: This subroutine restores the three global class variables CustomerNumber, CancelCustomerNumber, and CancelCustomerName from their corresponding Viewstate variables.

Lines 200 - 205: This subroutine saves three global class variables CustomerNumber, CancelCustomerNumber, and CancelCustomerName to their corresponding Viewstate variables.

Lines 207 - 212: This subroutine is called to perform server-side validation on the data entered by the user to update the record. While this routine only checks one value, it could easy be fanned out to check many. For each input value that needs to be checked, there should be a CustomValidator added to the form (in this validateCMCity is a custom validator for the txtCMCity TextBox). You could add whatever code necessary to check the control's value, including performing file IO if necessary (as you might do in the case of ensuring data integrity checks). This routine contrives a somewhat arbitrary test to ensure that no numeric values are present in the txtCMCity TextBox's Text property.

```
0154      BegFunc ReadRecord Type( *Boolean )
0155          // Read the customer record for display.
0156          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0157
0158          // Connect DB and open files.
0159          OpenData()
0160
0161          // Read the record--do not lock the record.
0162          Chain CustByNumber Key( CustomerNumber ) Access( *NoLock )
0163          If ( CustByNumber.IsFound )
0164              // Populate the UI if record is found.
0165              PopulateUIFromRecord()
0166          Else
0167              // Error handling code here.
0168          EndIf
0169
0170          // Close files and disconnect.
0171          CloseData()
0172
0173          // Assume the best for now!
0174          LeaveSr *True
0175      EndFunc
0176
0177      BegFunc RemoveNonNumericCharacters Type( *String )
0178          // Use a regular expression to remove all non-numeric characters
0179          // from an input string.
0180          DclSrParm Value Type( *String )
0181
0182          // [^0-9] is the regular expression that searches a string
0183          // for all occurrence of non-numeric values.
0184          LeaveSr Regex.Replace( Value, "[^0-9]", String.Empty )
0185      EndFunc
0186
0187      BegSr RemoveSessionVariables
0188          // Remove session variables.
0189          Session.Remove( "cancelnumber" )
0190          Session.Remove( "cancelname"   )
0191      EndSr
0192
0193      BegSr RestoreGlobalVariables
0194          // Load program global variables from viewstate variables.
0195          *This.CustomerNumber       = Viewstate[ "customernumber"       ].ToString()
0196          *This.CancelCustomerNumber = Viewstate[ "cancelcustomernumber" ].ToString()
0197          *This.CancelCustomerName   = Viewstate[ "cancelcustomername"   ].ToString()
0198      EndSr
0199
0200      BegSr SaveGlobalVariables
0201          // Save program variables to viewstate variables.
0202          Viewstate[ "customernumber"       ] = *This.CustomerNumber
0203          Viewstate[ "cancelcustomername"   ] = *This.CancelCustomerName
0204          Viewstate[ "cancelcustomernumber" ] = *This.CancelCustomerNumber
0205      EndSr
0206
0207      BegSr ValidateForm
0208          // Perform server-side custom validation.
0209          If ( RegEx.Match( txtCMCity.Text, "[0-9]" ).Success )
0210              validateCMCity.IsValid = *False
0211          EndIf
0212      EndSr
0213
```

Lines 214 - 236: Perform the actual Update operation on the customer number record.

Line 219:	Call OpenData() to connect the DB object and open the CustByName file.
Line 222:	Read the customer record for update with the customer number provided. Note the record is locked this time.
Line 223:	If the record was read successfully...
Line 225:	Populate the record format with values from the user interface.
Line 226:	Peform the Update operation.
Line 227:	If the record was not read successfully...
Line 228:	Put error handling here if the record was not read successfully.
Line 235:	Return the results of the update operation. In this case, true is the hardcoded return value. With rational error handling, the return value would be based on the success of the update operation.

```
0214      BegFunc UpdateRecord Type( *Boolean )
0215          // Update the customer record.
0216          DclSrParm CustomerNumber Like( Cust_CMCustNo )
0217
0218          // Connect DB and open files.
0219          OpenData()
0220
0221          // Read record for update.
0222          Chain CustByNumber Key( CustomerNumber )
0223          If ( CustByNumber.IsFound )
0224              // If record found, populate its fields and update it.
0225              PopulateRecordFromUI()
0226              Update CustByNumber
0227          Else
0228              // Error handling code here.
0229          EndIf
0230
0231          // Close files and disconnect.
0232          CloseData()
0233
0234          // Assume the best for now!
0235          LeaveSr *True
0236      EndFunc
0237  EndClass
```

This page is intentionally left blank.

Chapter 8

Data files used the examples in this book

File definition: Examples/CMastNewL1

This logical file is used to update a customer record.

```
Database Name.: *PUBLIC/DG NET Local
Library.......: Examples
File..........: CMastNewL1
File alias....: CustomerByNumber
Format........: RCMMastL1
Type..........: Simple
Base file.....: Examples/CMastNew
Description...: Cust Master by CustNo
Record length.: 151
Key length....: 5
Key field(s)..: CMCustNo

Field name           Data type   Length  Decimals  Description
--------------------------------------------------------------------------------
 CMCustNo            Packed           9        0    Cutomer Number
 CMName              Char            40             Customer Name
 CMAddr1             Char            35             Address Line 1
 CMCity              Char            30             City
 CMState             Char             2             State
 CMCntry             Char             2             Country Code
 CMPostCode          Char            10             Postal Code (zip)
 CMActive            Char             1             Active = 1, else 0
 CMFax               Packed          10        0    Fax Number
 CMPhone             Char            20             Main Phone
--------------------------------------------------------------------------------
```

File definition: Examples/CMastNewL2

This logical file is used to present customers in lists (by name and then number).

```
Database Name.: *PUBLIC/DG NET Local
Library.......: Examples
File..........: CMastNewL2
File alias....: CustomerByName
Format........: RCMMastL2
Type..........: Simple
Base file.....: Examples/CMastNew
Description...: Cust master by Name
Record length.: 151
Key length....: 45
Key field(s)..: CMName, CMCustNo
```

Field name	Data type	Length	Decimals	Description
CMCustNo	Packed	9	0	Cutomer Number
CMName	Char	40		Customer Name
CMAddr1	Char	35		Address Line 1
CMCity	Char	30		City
CMState	Char	2		State
CMCntry	Char	2		Country Code
CMPostCode	Char	10		Postal Code (zip)
CMActive	Char	1		Active = 1, else 0
CMFax	Packed	10	0	Fax Number
CMPhone	Char	20		Main Phone