



**Language Concepts Manual for
ASNA Visual RPG® 8.1 for Microsoft® Visual Studio® 2005**

Contents

Visual RPG Language Concepts.....	1
Arrays	1
Arrays Overview	1
Array Usage	2
Assemblies	3
Allocating Memory	3
Namespaces.....	5
Namespace Declarations	5
Namespace Concepts	6
Custom Attributes.....	6
Control Flow.....	7
Decision Structures	7
Loop Structures	8
Nested Control Statements	8
Data Types	8
Data Type Declaration	8
Elementary data types.....	9
Numeric Data Types	9
Integral Types	9
NonIntegral Types.....	9
*Char Data Type	9
Miscellaneous Data Types	10
Data Type Implementation.....	10
Values Types and Reference Types	10
Value Types vs. Reference Types.....	10
Data Types as Classes and Structures	11
Data Types Comparison.....	11
Declared Elements	12
Declared Element Characteristics	12
Lifetime	13
Static Variable Initialization	14
Declared Element Scope	15
Levels of Scope	15
Accessibility	16
Expressions.....	17
Calculating Numeric Values	17
Value Comparison	17
Boolean Expressions	18
Objects.....	19
What is an Object?.....	19
Relation of Objects to Each Other	19
The Basics of Working with Objects	20
Objects, Properties, Methods, and Events	20
Using Constructors.....	21
Babel Icons	Error! Bookmark not defined.

Operation Codes	21
Assignment Operation Codes.....	22
Declaration Operation Codes	23
Executable Operation Codes.....	24
Operators	24
Arithmetic operators	25
Arithmetic Operations	25
Division by Zero	25
Bitwise Operations.....	26
Bit Shift Operations	26
Comparison Operators.....	27
Comparing Strings.....	27
Comparing Objects	28
Concatenation Operators	28
Logical Operators.....	29
Strings	29
Manipulating Strings	30
Variables	33
Variable Declaration	33
Object Variables	34
Object Variable Declaration.....	35
Object Variable Assignment	35

Visual RPG Language Concepts

The following topics introduce the essential components of Visual RPG, an object-oriented programming language. After creating the interface for your application using forms and controls, you will need to write the code that defines the application's behavior. As with any modern programming language, Visual RPG supports a number of common programming constructs and language elements.

If you've programmed in other languages, much of the material covered in this section may seem familiar. While most of the constructs are similar to other languages, the event-driven nature of Visual RPG introduces some subtle differences.

If you're new to programming, the material in this section serves as an introduction to the basic building blocks for writing code. Once you understand the basics, you can create powerful applications using Visual RPG.

Arrays

When you use arrays, you can refer to multiple variables by the same name using a number called an *index* or to distinguish them from one another. Arrays can simplify and shorten your code, allowing you to create loops that deal efficiently with any number of elements.

Arrays Overview

Arrays allow you to refer to a series of variables by the same name and to use a number, called an *index* or to tell them apart. This helps you create shorter and simpler code in many situations, because you can set up loops that deal efficiently with any number of elements by using the index number.

Array Dimensions

An array can have one dimension or more than one. The *dimensionality* or *rank* corresponds to the number of subscripts used to identify an individual element. You can specify up to 32 dimensions, although more than three is extremely rare. Visual RPG allows you to specify the array dimensions by using the Rank keyword when using **BEGFUNC**, **BEGPROP**, **DCLARRAY** and **DCLSRPARRM**.

Array Size

Every dimension of an array has a nonzero length. The elements of the array are contiguous along each dimension from subscript 0 through the highest subscript of that dimension. All arrays are 0-based. Because Visual RPG allocates space for an array element corresponding to each index number, you should avoid declaring any dimension of an array larger than necessary.

A **zero-length array** is an array with no elements. It is sometimes necessary to specify such an array, for example when using Windows Forms. To do this, you declare one of the array's dimensions to be -1. The array is then empty, but it still exists. Therefore, a variable pointing to the array is not equal to *Nothing.

Array Class

All arrays inherit from the Array class in the System namespace, and you can access the methods and properties of System.Array on any array.

Array Type (*Object)

Arrays are objects in Visual RPG, so every array type is an individual reference type. This has the following implications:

- No two array variables are considered to be of the same data type unless they have the same rank and element data type.
- When you assign one array variable to another, only the pointer is copied.
- An array variable holds a pointer to the data constituting the elements and the rank and length information.

Array Element Type

An array declaration specifies a data type, and all its elements must be of that type. When the data type is Object, the individual elements can contain different kinds of data (objects, strings, numbers, and so on).

You can also declare an array that contains other arrays as its elements. In this case, the constituent arrays must all have the same element data type, which you specify in the declaration. An array of arrays is called a *jagged array* because the constituent arrays do not have to have the same sizes.

Array Operation Codes/Operators

The following are the operation codes/operators that operate on every element of the array when a single dimension fixed sized array name is used as the result field of a command or assignment expression:

- + (Unary and binary plus), - (unary and binary minus), * (multiply), / (divide), *Not, *And, *Or operators:

Example: `array1 = array2 + 5 - field3`

// 5 is added and field3 is subtracted to every element of array2 and the result is assigned to every corresponding element of array2. If array1 and array2 have different element counts, the operation is done using the minimum count.

- ADD, DIV, MULT, SQRT, SUB, ZADD **and** ZSUB.
- EXTRACT, ADDDUR, and SUBDUR commands.
- MOVE commands.

Every array that is used by stating its name with these operators will participate in the 'array indexing' of the expression. Array properties and functions that return arrays are not automatically indexed.

Array Usage

Because automatic array indexing is a compile-time operation, it applies only when the resulting field is an array declared using the DIM keyword, and does NOT apply when resulting fields are arrays that use the RANK keyword. This is particularly important on assignments of the form `array1 = <some expression>`.

- If array1 is a DIM-declared array, then the assignment expression WILL be indexed and follow the RPG rules (shorter array, etc). Any RANK-declared array in the expression will participate in indexing, and the user must be sure that the rank array contains sufficient elements, or a run-time error will occur.
- If array1 is a RANK-declared array, then no automatic indexing will take place in expressions that have arrays as operands. E.g. `myArr1 = myArr2 * myArr3`

If you want to use arrays in expressions, then those arrays must be RPG arrays (DIM arrays).

For the commands that operate on arrays, the array in the RESULT field must be a DIM array.

Examples

// Example 1

```
dimArray1 = dimArray2 + rankArray1
```

becomes...

```
DO 0 <shorter length of dimArray1 and dimArray2 minus 1> Index
```

```
dimArray1[ Index ] = dimArray2[ Index ] + rankArray1[ Index ]
```

```
ENDDO
```

// remains the same, no code changes

```
rankArray1 = dimArray1
```

// this generates a compiler error

```

rankArray1 = dimArray1 + dimArray2
// Example 4
ZADD 3 dimArray1
becomes...
DO 0 <length of dimArray1 minus 1> Index
ZADD 3 dimArray1[ Index ]
ENDDO
// This generates a compiler error
ZADD 4 rankArray1

```

Assemblies

Assemblies form the building blocks of the .NET Framework, and take the form of an executable (.exe) file or dynamic link library (.dll) file. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

With Visual RPG .NET, you use the contents of assemblies, and add references to them. What makes assemblies different from .exe or .dll files in earlier versions of Windows, however, is that they contain all the information you would find in a type library, plus information about everything else necessary to use the application or component.

Assembly Manifest

Within every assembly is an *assembly manifest*. Similar to a table of contents, the assembly manifest contains the following:

- A file table describing all the other files that make up the assembly, including, for example, any other assemblies you created that your .exe or .dll file relies on, or even bitmap or Readme files.
- The assembly's identity (its name and version).
- An *assembly reference list*, which is a list of all external dependencies — .dlls or other files your application needs that may have been created by someone else. Assembly references contain references to both global and private objects. Global objects reside in the global assembly cache, an area available to other applications, somewhat like the System32 directory. The ASNA.Visual RPG namespace is an example of an assembly in the global assembly cache. Private objects must be in a directory at either the same level as or below the directory in which your application is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications you create with Visual RPG .NET do not rely on registry values to function properly. Assemblies reduce DLL conflicts and make your applications more reliable and easier to deploy.

Allocating Memory

Visual Studio's memory management is one of the services that the common language runtime provides during Managed Execution. The common language runtime's garbage collector manages the allocation and release of memory for an application. What this means for developers is that you do not have to write code to perform memory management tasks when you develop managed applications. Automatic memory management can eliminate common problems, such as forgetting to release an object and causing a memory leak, or attempting to access memory for an object that has already been released. This section describes how the garbage collector allocates and releases memory in .NET.

When you initialize a new process, the runtime reserves a contiguous region of address space for the process. This reserved address space is called the managed heap. The managed heap maintains a pointer to the address where the next object in the heap will be allocated. Initially, this pointer is set to the managed heap's base address. All reference types are allocated on the managed heap. When an application creates the first reference type, memory is allocated for the type at the base address of the managed heap.

When the application creates the next object, the garbage collector allocates memory for it in the address space immediately following the first object. As long as address space is available, the garbage collector continues to allocate space for new objects in this manner.

Allocating memory from the managed heap is faster than unmanaged memory allocation. Because the runtime allocates memory for an object by adding a value to a pointer, it is almost as fast as allocating memory from the stack. In addition, because new objects that are allocated consecutively are stored contiguously in the managed heap, an application can access the objects very quickly.

Releasing Memory

The garbage collector's optimizing engine determines the best time to perform a collection based on the allocations being made. When the garbage collector performs a collection, it releases the memory for objects that are no longer being used by the application. It determines which objects are no longer being used by examining the application's roots. Every application has a set of roots. Each root either refers to an object on the managed heap or is set to null. An application's roots include global and static object pointers, local variables and reference object parameters on a thread's stack, and CPU registers. The garbage collector has access to the list of active roots that the just-in-time compiler and the runtime maintain. Using this list, it examines an application's roots, and in the process creates a graph that contains all the objects that are reachable from the roots.

Objects that are not in the graph are unreachable from the application's roots. The garbage collector considers unreachable objects garbage and will release the memory allocated for them. During a collection, the garbage collector examines the managed heap, looking for the blocks of address space occupied by unreachable objects. As it discovers each unreachable object, it uses a memory-copying function to compact the reachable objects in memory, freeing up the blocks of address spaces allocated to unreachable objects. Once the memory for the reachable objects has been compacted, the garbage collector makes the necessary pointer corrections so that the application's roots point to the objects in their new locations. It also positions the managed heap's pointer after the last reachable object. Note that memory is compacted only if a collection discovers a significant number of unreachable objects. If all the objects in the managed heap survive a collection, then there is no need for memory compaction.

To improve performance, the runtime allocates memory for large objects in a separate heap. The garbage collector automatically releases the memory for large objects. However, to avoid moving large objects in memory, this memory is not compacted.

Generations and Performance

To optimize the performance of the garbage collector, the managed heap is divided into three generations: 0, 1, and 2. The runtime's garbage collection algorithm is based on several generalizations that the computer software industry has discovered to be true by experimenting with garbage collection schemes. First, it is faster to compact the memory for a portion of the managed heap than for the entire managed heap. Secondly, newer objects will have shorter lifetimes and older objects will have longer lifetimes. Lastly, newer objects tend to be related to each other and accessed by the application around the same time.

The runtime's garbage collector stores new objects in generation 0. Objects created early in the application's lifetime that survive collections are promoted and stored in generations 1 and 2. The process of object promotion is described later in this topic. Because it is faster to compact a portion of the managed heap than the entire heap, this scheme allows the garbage collector to release the memory in a specific generation rather than release the memory for the entire managed heap each time it performs a collection.

In reality, the garbage collector performs a collection when generation 0 is full. If an application attempts to create a new object when generation 0 is full, the garbage collector discovers that there is no address space remaining in generation 0 to allocate for the object. The garbage collector performs a collection in an attempt to free address space in generation 0 for the object. The garbage collector starts by examining the objects in generation 0 rather than all objects in the managed heap. This is the most efficient approach, because new objects tend to have short lifetimes, and it is expected that many of the objects in generation 0 will no longer be in use by the application when a collection is performed. In addition, a collection of generation 0 alone often reclaims enough memory to allow the application to continue creating new objects.

After the garbage collector performs a collection of generation 0, it compacts the memory for the reachable objects as explained in Releasing Memory earlier in this topic. The garbage collector then promotes these objects and considers this portion of the managed heap generation 1. Because objects that survive collections tend to have longer lifetimes, it makes sense to promote them to a higher generation. As a result, the garbage collector does not have to reexamine the objects in generations 1 and 2 each time it performs a collection of generation 0.

After the garbage collector performs its first collection of generation 0 and promotes the reachable objects to generation 1, it considers the remainder of the managed heap generation 0. It continues to allocate memory for new objects in generation 0 until generation 0 is full and it is necessary to perform another collection. At this point, the garbage collector's optimizing engine determines whether it is necessary to examine the objects in older generations. For example, if a collection of generation 0 does not reclaim enough memory for the application to successfully complete its attempt to create a new object, the garbage collector can perform a collection of generation 1, then generation 0. If this does not reclaim enough memory, the garbage collector can perform a collection of generations 2, 1, and 0. After each collection, the garbage collector compacts the reachable objects in generation 0 and promotes them to generation 1. Objects in generation 1 that survive collections are promoted to generation 2. Because the garbage collector supports only three generations, objects in generation 2 that survive a collection remain in generation 2 until they are determined to be unreachable in a future collection.

Releasing Memory for Unmanaged Resources

For the majority of the objects that your application creates, you can rely on the garbage collector to automatically perform the necessary memory management tasks. However, unmanaged resources require explicit cleanup. The most common type of unmanaged resource is an object that wraps an operating system resource, such as a file handle, window handle, or network connection. Although the garbage collector is able to track the lifetime of a managed object that encapsulates an unmanaged resource, it does not have specific knowledge about how to clean up the resource. When you create an object that encapsulates an unmanaged resource, it is recommended that you provide the necessary code to clean up the unmanaged resource in a public `Dispose` method. By providing a `Dispose` method, you enable users of your object to explicitly free its memory when they are finished with the object. When you use an object that encapsulates an unmanaged resource, you should be aware of `Dispose` and call it as necessary.

Namespaces

Due to the .NET framework, Visual RPG .NET programs are organized using namespaces, both to internally organize a program, as well as to organize the way program elements are exposed to other programs.

Unlike other entities, namespaces are open-ended, and may be declared multiple times within the same program and may be declared across many programs, with each declaration contributing members to the same namespace.

Use the **DCLNAMESPACE** command to declare a particular namespace in code.

There is a global namespace that has no name and whose nested namespaces and types can always be accessed without qualification. The scope of a namespace member declared in the global namespace is the entire program text.

Namespace Declarations

A namespace declaration (`DCLNAMESPACE`) consists of the name of the namespace member. If the namespace name is qualified, the namespace declaration is treated as if it is lexically nested within namespace declarations corresponding to each name in the qualified name.

When dealing with the members of a namespace, it is not important where a particular member is declared. If two programs define an entity with the same name in the same namespace, attempting to resolve the name in the namespace causes an ambiguity error.

Namespaces are by definition `*Public`, so a namespace declaration cannot include any access modifiers.

Note: The project itself defines a root namespace, so any namespace defined within a file is nested within it.

Namespace Concepts

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

For example, Visual RPG .Net defines the `ListBox` class in the `System.Windows.Forms` namespace. The following code fragment shows how to declare a variable using the fully qualified name for this class:

```
DCLNAMESPACE mynamespace(System.Windows.Forms.ListBox)
```

By default, every executable file you create with Visual RPG .Net contains a namespace with the same name as your project. For example, if you define an object within a project named `MyProject`, the executable file, `MyProject.exe`, contains a namespace called `MyNamespace`.

Multiple assemblies can use the same namespace. Visual RPG .Net treats them as a single set of names. For example, you can define classes for a namespace called `MyNameSpace` in an assembly named `MyAssembly1`, and define additional classes for the same namespace from an assembly named `Assembly2`.

Fully Qualified Names

Fully qualified names are object references that are prefixed with the name of the namespace where the object is defined. You can use objects defined in other projects if you create a reference to the class (by choosing `Add Reference` from the `Project` menu) and then use the fully qualified name for the object in your code. The following code fragment shows how to use the fully qualified name for an object from another project's namespace:

```
DCLFLD MyField *New (MyProject.Form1.ListBox() )
```

If you attempt to use a `class` without fully qualifying it, Visual RPG .Net produces an error stating that the name `class1` is ambiguous.

Namespace Level Statements

Within a namespace, you can define items such as programs, interfaces, classes, delegates, enumerations, structures, and other namespaces. You cannot define items such as properties, procedures, variables and events at the namespace level. These items must be declared within containers such as modules, structures, or classes.

Custom Attributes

Custom Attributes are descriptive tags that provide additional information about programming elements such as classes, fields, methods, and properties. Other applications, such as the Visual RPG compiler or the Visual Studio designers, can refer to the extra information in attributes to determine how these items can be used at design-time and run-time.

Attributes and Metadata

Attributes are saved with the *metadata* of Visual RPG assemblies. Metadata is information that describes every element managed by the system runtime. This includes information required for debugging and garbage collection, as well as security attributes, marshaling data, extended class and member definitions, version binding, and any other information the runtime requires.

With attributes, you specify the metadata in much the same way as you use special values like `*Public` and `*Private` to provide information about Access levels. However, unlike special values, most attributes are not Visual RPG-specific. Using attributes, you can extend the capabilities of the Visual RPG language without requiring changes to the compiler.

Functionality and Capabilities of Attributes

Some key points about attributes include:

- You can apply one or more attributes to entire assemblies, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.

- Attributes fall into two groups: those that are used by the .NET Framework and Visual RPG, and custom attributes that are meaningful only to specific applications.
- While the .NET Framework contains many useful attributes, you can define your own custom attributes if necessary.

The Attributes Keyword

The following commands have a new optional keyword call Attributes which allow you to provide one or more attributes on the object being defined by the command. To specify more than one attribute, simply separate them with commas.

- BEGCLASS
- BEGCONSTRUCTOR
- BEGFUNC
- BEGPROP
- BEGSR
- DCLARRAY
- DCLEVENT
- DCLFLD

You can also add attributes to the assembly by using the new command:

SetAssemblyAttribute Attribute(attr)

To specify more than one attribute for the assembly use multiple SetAssemblyAttribute commands. This command can be coded in any of you classes but must be come before the BEGCLASS and after any USING statements.

It is common to create a dedicated source file called AssemblyInfo.vr where all assembly attributes are grouped. This file usually lacks a class definition, i.e.: it contains no BegClass whatsoever.

Control Flow

Left unregulated, a program proceeds through its commands from beginning to end. Some very simple programs can be written with only this unidirectional flow. However, most of the power and utility of any programming language comes from the ability to change execution order with control commands and loops.

Control-flow commands allow you to regulate the flow of your program's execution. Using control commands, you can write Visual RPG code that makes decisions and repeats actions.

Decision Structures

Decision structures allow you to execute one or more lines of code based upon conditions tested, and then perform different operations depending on the results of that test. You can test for a condition being true or false, for specific values of an expression, or for various exceptions generated when you execute statements. The decision structures supported by Visual RPG include:

- IF...then...ELSE process a group of commands when a comparison expression evaluation is True(IF), or False(ELSE). The comparison expression is evaluated at the end of the commands, and if the expression is true, the group of commands are executed again. An ENDIF or END command is required to end the group of commands.
- SELECT conditionally processes one of several alternative sequences of operations. It consists of a SELECT statement, zero or more WHEN groups, an optional OTHER group, and terminates with an END or ENDSL statement. You can also use CASE, to specify an comparison expression to select a subroutine for processing when the condition evaluates True.

- **TRY..CATCH..FINALLY** provides a way to trap possible errors that may occur in code segments, while still running the code. The TRY statements are executed first, any exceptions are trapped with the CATCH commands, and FINALLY commands are executed when no exceptions are trapped. An ENDRY command is required to end the group of commands.

Loop Structures

Loop structures allow you to execute one or more lines of code repetitively. You can repeat the commands until a condition is true, until a condition is false, a specified number of times, or once for each object in a collection. The loop structures supported by Visual RPG include:

- **Do** begins a group of operations and indicates the number of times the group will be processed. An **ENDDO** statement marks the end of the group.
- **DOWHILE** executes a block of commands while a comparison expression evaluates true. An **ENDDO** statement marks the end of the group.
- **DUNTIL** executes a block of commands until a comparison expression evaluates true. An **ENDDO** statement marks the end of the group.
- **FOR** executes a block of commands a specified number of times, until the index value is greater than or less than the limit value, depending on the keyword values. After the number of iteration has been completed, control is transferred to the next statement following the FOR/ENDFOR block.
- **FOREACH** executes a block of commands for each element in an array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the FOREACH/ENDFOR block.

Nested Control Statements

You can place control commands inside other control commands, for example an If...Then...Else block within another loop. A control command placed inside another control command is said to be *nested*.

Control commands in Visual RPG can be nested to as many levels as you want. It is common practice to make nested decision structures and loops more readable by indenting the body of each one. See the individual commands within the **Visual RPG Language Reference** for specific nesting structures.

Data Types

The **data type** of a programming element refers to what kind of data it can hold and how that data is stored. Data types apply to all values that can be stored in computer memory or participate in the evaluation of an expression. Every variable, literal, constant, property, procedure argument, and procedure return value has a data type.

A **declared data type** is specified as part of the declaration of a programming element. Unless you use typeless programming, you must declare data types for all your programming elements.

This section describes the declaration and use of Visual RPG data types.

Data Type Declaration

Every programming element that can hold data has a declared data type. The following list shows how to specify data types of various elements:

- Variable — In a declaration statement.
- Literal — With a literal type character.
- Constant — In a declaration statement.
- Property — In a procedure declaration statement.
- Procedure argument — In the procedure declaration.

- Procedure return value — In the procedure declaration.

Elementary data types

Visual RPG .NET supplies a set of pre-defined data types that you can use for many of your programming elements. This section describes these types and how to use them.

Numeric Data Types

Visual RPG supplies several numeric data types for handling numbers in various representations. Integral types represent only whole numbers, and nonintegral types represent numbers with both integer and fractional parts.

Integral Types

Integral data types are those that represent only whole numbers. If a variable always stores whole numbers rather than numbers with a fractional amount, declare it as one of these types.

The unsigned integral type is `*Byte` (8-bit). If a variable contains binary data, or data of unknown nature, declare as this type. If more than eight bits of data are needed, declare the variable as an array of `*Byte` elements.

Arithmetic operations are faster with integral types than with other data types. They are fastest with the `*Integer` type in Visual RPG.

Since `*Byte` is an unsigned type with a range of 0-255, it cannot represent a negative number. If you use the unary minus (-) operator on an expression that evaluates to type `*Byte`, Visual RPG converts the expression to `*Integer` first.

If you try to set a variable of an integral type to a number outside the range for that type, an error occurs. If you try to set it to a fraction, the number is rounded.

Binary data stored in `*Byte` variables and arrays is preserved during format conversions. You should not use a `*String` variable for binary data, because its contents can be corrupted during conversion between ANSI and Unicode formats. Such conversion can happen automatically when Visual RPG reads or writes files, or when it calls DLLs, methods, and properties.

NonIntegral Types

Nonintegral data types are those that represent numbers with both integer and fractional parts. The nonintegral nonIntegral types are `*Float`.

Floating point numbers can be subject to rounding errors. Floating point types support fewer significant digits than `*Integer` but can represent values of greater magnitude.

Floating point values can be expressed as *mmmEeee*, in which *mmm* is the mantissa (the significant digits) and *eee* is the exponent (a power of 10).

*Char Data Type

The `*Char` data type is a finite-length string of Unicode text characters that compose an RPG field. Internally, this field is characterized as a string, but, as a field, the length of the string is always the exact length of the `*Char` field definition.

For example:

```
dclFld CustName type( *char ) Len( 35 )
dclFld myStr type( *string )
myStr = CustName.Trim() // trim white space
```

Miscellaneous Data Types

Visual RPG supplies several data types that are not oriented toward numbers or characters. Instead, they deal with specialized data such as yes/no and date/time values.

Data Type Implementation

The Visual RPG data types can be classified according to whether a variable of a particular type stores its own data or a pointer to the data. This classification affects the way a data type is implemented.

Values Types and Reference Types

Although value types and reference types can be similar in terms of declaration syntax and usage, their semantics are distinct.

Reference types are stored on the run-time heap; only being accessed through a reference to that storage. This allows the garbage collector to track outstanding references to a particular instance and free the instance when no references remain. A variable of reference type always contains a reference to a value of that type or a null reference.

- A null reference refers to nothing; it is invalid to do anything with a null reference except assign it.
- Assignment to a variable of a reference type creates a copy of the reference, not a copy of the value being referenced.

Value types are stored directly on the stack, either within an array or within another type. When the location containing a value type instance is destroyed, the value type instance is also destroyed. Value types are always accessed directly; it is not possible to create a reference to a value type. Prohibiting such a reference makes it impossible to refer to a value class instance that has been destroyed. A variable of a value type always contains a value of that type.

Unlike reference types, the value of a value type cannot be a null reference, nor can it reference an object of a more derived type. Assignment to a variable of a value type creates a copy of the value being assigned.

Value Types vs. Reference Types

AVR for .NET references value types and reference types.

Value types include simple types (e.g., char, int, and float), enum types, and structure types. Reference types include class types, interface types, delegate types, and array types. Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store references to objects.

With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

In AVR for .NET, pretty much all types (*) are value types, except: *object, *string, *form, and in 7.1 enumerations (BeginInit).

- Fields declared with a value type have a default value which depends on the particular type (like 0 for numerics or blanks for *char).
- Fields declared with a reference type all have the same default value which is *Nothing. **Therefore they must be initialized before they can be used.** You can test for *nothing in conditional expressions (eg. IF (mystring = *nothing) ...)

Data Types as Classes and Structures

Every elementary data type in Visual RPG is supported by a structure or a class in the System namespace. The compiler uses each data type keyword as an alias for the underlying structure or class. For example, declaring a variable with the reserved word Byte is the same as declaring it with the fully qualified structure name System.Byte.

In the .NET Framework, a structure is a value type and a class is a reference type. For this reason, value types such as *Char and *Integer are supported by .NET structures, while reference types such as *Object and *String are supported by .NET classes.

Since every reference type represents an underlying .NET class, you must use the *New special value when you initialize it.

You can also use the *New special value to initialize a value type. This is particularly useful if the type has a constructor that takes parameters.

Data Types Comparison

The following table shows the Visual RPG data types, their supporting .NET framework type, and the equivalent C# and Visual Basic data type. They are listed by hierarchical view of data types. Additional information can be found in the Visual RPG Language Reference links within the table and the 'Related Sections'

Visual RPG type	.NET Framework Type	C#	Visual Basic
REFERENCE TYPES			
*String	System.String	string	String
*Char	System.String	N/A	N/A
*Object	System.Object	System.Object	System.Object
*Form	System.Windows.Forms.Form	System.Windows.Forms.Form	System.Windows.Forms.Form
VALUE TYPES:			
Structure Types:			
*Date	System.DateTime	datetime	DateTime
*Time	System.DateTime	datetime	DateTime
*TimeStamp	System.DateTime	datetime	DateTime
Simple Types:			
Logical:			
*Boolean	System.Boolean	bool	Boolean

*Ind	System.Char	N/A	N/A
Numeric Decimal:			
*Binary	System.Decimal	N/A	N/A
*Decimal	System.Decimal	decimal	Decimal
*Packed	System.Decimal	decimal	Decimal
*Zoned	System.Decimal	decimal	Decimal
Numeric Floating Point:			
*Float4	System.Float32	System.Float32	System.Float32
*Float8	System.Float64	System.Float64	System.Float64
Numeric Integral:			
*Byte	System.Byte	byte	Byte
*Integer2	System.Int16	short	Short
*Integer4	System.Int32	int	Integer
Integer8	System.Int64	long	Long
*OneChar	System.Char	char	Char

Remarks

To be completely precise, all values types are structures derived from System.Value. The distinction this chart draws between *Structure Types* and *Simple Types* is that simple types focus on providing one data element (their value), whereas the date/time structures provide several data elements as sub elements of the date/time values.

Declared Elements

A *declared element* is a programming element that is defined in a declaration statement. Declared elements include variables, arrays, constants, databases, properties, methods, events, files, and fields. See Declaration Operation Codes for a listing.

Declared Element Characteristics

Every *declared element* (a programming element that is defined in a declaration statement) has one or more of the following characteristics associated with it:

- Data type - (Type keyword) The values the element can hold, and how those values are stored, such as *BINARY, *BOOLEAN, *BYTE, *CHAR, *DATE, *DECIMAL | *FLOAT, *IND, *INTEGER, *OBJECT, *ONECHAR, *PACKED, *STRING, *TIME, *TIMESTAMP, or *ZONED.
- Lifetime - (Static keyword) The period the declared field will remain in existence and retain their latest values after termination of the procedure in which they are declared.

- Scope - (Access keyword) The type of access to the file, such as (PRIVATE, *PUBLIC, *PROTECTED, or *INTERNAL).
- Accessibility (Shared keyword) The permission for code to use the element.

The following table shows the declared elements and the characteristics that apply to each one.

Element	Data Type (Type)	Lifetime (Static)	Scope (Access)	Accessibility (Shared)
Class	No	No	Yes	Yes
Constant	Yes	No	Yes	Yes
Enumeration	Yes	No	Yes	Yes
Event	No	No	Yes	Yes
Property	Yes	No	Yes	Yes
Method	No	No	Yes	Yes
Procedure	No	No	Yes	Yes
Function return	Yes	Yes	Yes	No
Delegate	No	No	Yes	Yes
Interface	No	No	Yes	Yes
Variable	Yes	Yes	Yes	Yes

Visibility

Visibility results from a combination of lifetime, scope, and accessibility. A programming element can be said to be *visible* from a block of code when all three of the following conditions apply:

- The element is within its lifetime (Static) — that is, it has been created and not yet terminated.
- The access of the element includes the block of code referring to it.
- The element's sharing accessibility permits the code to use it.

Lifetime

The *lifetime* of a declared element is the period of time during which it is available for use. Variables are the only elements that have lifetime; for this purpose, the compiler treats procedure arguments and function returns as special cases of variables. The lifetime of a variable represents the period of time during which it can hold a value. Its value can change over its lifetime, but it always holds some value.

Beginning of Lifetime

A local variable's lifetime begins when execution enters the procedure in which it is declared. Every local variable is initialized to the default value for its data type as soon as the procedure begins execution. Numeric variables (including *Byte and *Char) are initialized to zero, *Date variables to midnight of January 1 of the year 1, Boolean variables to False, and reference type variables (including strings, arrays, and Object) to *Nothing.

Each member of a structure variable is initialized as if it were a separate variable. Similarly, each element of an array variable is initialized individually.

If a variable is declared with an initializer, the specified value is assigned to the variable when its declaration statement is executed, as shown in the following example:

```
DCLFLD Name( MyField ) Type( *Integer )  
MyField = 45 //MyField had previously been initialized to 0.
```

Variables declared within a block inside a procedure are initialized to their default values on entry to the procedure. These initializations take effect whether or not the block is ever executed.

End of Lifetime

When a procedure terminates, the values of its local variables are not preserved, and the memory used by the local elements is reclaimed. The next time the procedure is executed, all its local elements are created afresh and the local variables are initialized.

When an instance of a class or structure terminates, its nonshared variables lose their values. Each new instance of the class or structure creates all its nonshared elements and initializes the nonshared variables. Shared elements are preserved until your application stops running.

Different Lifetimes

A variable declared at module level typically exists for the entire time your application is running. A nonshared variable declared in a class or structure exists as a separate copy for each instance of the class or structure in which it is declared; each such variable has the same lifetime as its instance. However, a shared variable has only a single lifetime, which lasts for the entire time your application is running.

Local variables or arrays declared with **DCLFLD** or **DCLARRAY** exist only while the procedure in which they are declared is executing. This applies also to that procedure's arguments and to any function return. However, if that procedure calls other procedures, the local variables retain their values while the called procedures are running.

Extension of Lifetime

If a local variable or array is declared with the `Static(*Yes)` keyword, its lifetime is longer than the execution time of the procedure in which it is declared. If the procedure is inside a module, the static variable survives as long as your application continues running.

If a static variable is declared in a procedure inside a class, the variable's lifetime depends on whether the procedure is shared. If the procedure is declared with the `Shared(*Yes)` keyword, the variable's lifetime continues until your application terminates. If the procedure is nonshared, its static variables are instance members of the class, and their lifetimes are the same as that of the class instance.

Static Variable Initialization

You can initialize the value of a static local variable as part of its declaration. If you declare an array to be `Static`, you can initialize its Rank (number of dimensions), the length of each dimension, and the values of the individual elements.

You can declare static variables with the same name in more than one procedure. If you do this, the Visual RPG compiler considers each such variable to be a separate element. The initialization of one of these variables does not affect the values of the others. The same applies if you define a procedure with a set of overloads and declare a static variable with the same name in each overload.

You can declare a static local variable in a class, that is, inside a procedure in that class. However, you cannot declare a static local variable in a structure.

Declared Element Scope

The *scope* of a declared element is the set of all code that can refer to it without qualifying its name. An element can have *scope* at one of the following levels:

- Block access — available only within the code block in which it is declared.
- Procedure access — available only within the procedure in which it is declared.
- Program access — available to all code within the program, or class in which it is declared.
- Namespace access — available to all code in the namespace.

These levels of access progress from the narrowest (block) to the widest (namespace), where *narrowest scope* means the smallest set of code that can refer to the element without qualification.

You specify the scope of an element when you declare it depending on the following factors:

- The region (block, procedure, module, class, or structure) in which you declare the element.
- The namespace containing the element's declaration.
- The accessibility you declare for the element.

Use care when you define variables with the same name but different access, because doing so can lead to unexpected results.

Levels of Scope

A programming element is available throughout the region in which you declare it. All code in the same region can refer to the element without qualifying its name.

Namespace Scope

If you declare an element at program level using `Access (*Internal)` or `Access(*Public)`, it becomes available to all procedures throughout the namespace in which the element is declared.

Namespace access includes nested namespaces. An element available from within a namespace is also available from within any namespace nested inside that namespace.

If your project does not contain any `DCLNAMESPACE` commands, everything in the project is in the same namespace. In this case, namespace access can be thought of as project access. `*Public` elements in a program, class, or structure are also available to any project that references their project.

Program Scope

The single term *program level* applies equally to programs, classes, and structures. You can declare elements at this level by placing the declaration statement outside of any procedure or block within the program, class, or structure.

When you make a declaration at the program level, the accessibility you choose determines the access. The namespace that contains the program, class, or structure also affects the access.

Elements for which you declare `Access (*Private)` are available for reference to every procedure in that program, but not to any code in a different program. The `DCLFLD` statement at program level defaults to `Access (*Private)` if you do not use any accessibility keywords. However, it's better to make the access and accessibility more obvious by coding the `Access(*Private)` option in the `DCLFLD` command.

Procedure Scope

An element declared within a procedure is not available outside that procedure. Only the procedure that contains the declaration can use it. Elements at this level are also known as *local* elements. You declare them with the `DCLFLD` statement, with or without the `Static` keyword.

Procedure and block access are closely related. If an element is declared inside a procedure but outside any block within that procedure, the element can be thought of as having block access, where the block is the entire procedure.

Note All local elements, even if they are Static(*YES)variables, are private to the procedure in which they appear. You cannot declare an element using Access(*Public) within a procedure.

Block Scope

A set of statements terminated by an End, Else command is called a block, for example an If...Else...EndIf block. An element declared within a block can be used only within that block.

Note Even if the access of an element is limited to a block, its lifetime is still that of the entire procedure. If you enter the block more than once during the procedure, a block variable retains its previous value. It is wise to initialize the variable each time to avoid unexpected results in such a case.

Accessibility

The *accessibility* of a declared element is the ability to use it, that is, the permission for code to read it or write to it. This is determined not only by how you declare the element itself, but also by the accessibility of the element's container. If the containing element is not accessible, none of its contained elements are accessible, even those declared as Access (*Public). For example, a *Public variable in a *Private structure is accessible from inside the class containing the structure, but not from outside that class.

Public

The Access (*Public) keyword in the DCLFLD statement declares elements to be accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project. The following code shows a sample Public declaration:

```
DclFld MyField Type( *String ) Len (4) Access( *Public )
```

You can use *Public only at module, namespace, or file level. This means you can declare a public element in a source file or inside a class, but not within a procedure.

Protected

The Access(*Protected) keyword in DCLFLD declares elements to be accessible only from within the same class, or from a class derived from this class. The following code shows a sample Protected declaration:

```
DclFld MyField Type( *String ) Len (4) Access( *Protected )
```

You can use Protected only at class level, and only when declaring a member of a class.

Internal

The Access (*Internal) keyword in the Dim statement declares elements to be accessible from within the same project, but not from outside the project. The following code shows a sample *Internal declaration:

```
DclFld MyField Type( *String ) Len (4) Access( *Protected )
```

You can use *Internal only at namespace, or file level. This means you can declare a friend element in a source file or inside a class, but not within a procedure.

Private

The Access (*Private) keyword in DCLFLD declares elements to be accessible only from within the same module, class, or structure. The following code shows a sample *Private declaration:

```
DclFld MyField Type( *String ) Len (4) Access( *Protected )
```

You can use *Private only at namespace or file level. This means you can declare a private element in a source file or inside a class, but not within a procedure.

Expressions

An expression can be thought of as a series of value-representing elements, separated by operators, that yields a new value. Examples of such value-representing elements are literals, variables, other expressions, and function calls. The operators act upon the value-representing elements by performing calculations, comparisons, or other operations. The end result of the expression represents a value, which can be either numeric, *Boolean, or *String.

Calculating Numeric Values

Numeric values can be calculated through the use of numeric calculating numeric values. A numeric expression is an expression that contains literals and variables representing numeric values, and operators that act upon those values. The following is an example of a numeric expression:

```
4 * (67 + x) // For the purposes of this example, x = 2.
```

This expression evaluates to 276, which is the value of 67 plus 2, multiplied by 4.

If the numeric expression contains more than one operator, the order in which they are evaluated is determined by the rules of operator precedence. To override the rules of operator precedence, you enclose calculating numeric values in parentheses, as in the above example; such calculating numeric values are evaluated first. For more information, see **Operator Precedence in Visual RPG**.

You can use an assignment statement to assign the value represented by this expression to another variable, as in this example:

```
DCLFLD x Type (*Integer) Len (4)
  x = 2
DCLFLD y Type (*Integer) Len (4)
  y = 4 * (67 + x)
```

Here, the value of the expression on the right side of the operator is assigned to the variable `y` on the left side of the operator, so `y` evaluates to 276. For more information on assignment, see **Assignment Statements**.

Value Comparison

Comparison operators can be used to construct expressions that compare the values of numeric variables. These expressions return a Boolean value based on whether the comparison is true or false. Examples of such an expression would be:

```
88 > 76
34 > 55
```

The first expression evaluates to True, because 88 is greater than 76. The second example evaluates to False, because 34 is not greater than 55.

You can compare any two numeric expressions in this fashion. The expressions you compare can themselves be complex expressions, as in this example:

```
x / 55 * (y + 11) >= System.Math.Sqrt(z) / (p - (x * 16))
```

This complex expression includes literals, variables, and function calls. The expressions on both sides of the comparison operator are evaluated, and the resulting values are then compared using the `>=` comparison operator. If the value of the expression on the left side is greater than or equal to the value of the expression on the right, the entire expression evaluates to True; otherwise, it evaluates to False.

Expressions that compare values are most commonly used in If...Then statements, as in the following example:

```
If x > 30 Then
  // Insert code to be executed if x is greater than 30.
```

```
Else
  // Insert code to be executed if x is less than or equal to 30.
End If
```

The = sign is a comparison operator as well as an assignment operator. When used as a comparison operator, it evaluates whether the value on the left is equal to the value on the right, as shown in the following example:

```
If x = 30
  Then // Insert code to continue
program. EndIf
```

You can also use a comparison expression anywhere that a Boolean value is needed such as in an If, While, Loop, ElseIf command or when assigning to or passing a value to a Boolean variable. In the following example, the value returned by the comparison expression is assigned to a Boolean variable:

```
DclFld x *As (*Boolean)
x = 55 < 30 // x is assigned False.
```

Boolean Expressions

A Boolean expression is an expression that evaluates to a Boolean value. Boolean expressions can take several forms. The simplest is the direct comparison of the value of a Boolean variable to a Boolean literal, as shown in this example:

```
If x = True //Compares x to True
  // Insert code to execute if x= True.
Else // Insert code to execute if x= *alse
EndIf
```

Note that the assignment statement `x = True` looks the same as the expression in the example, but it performs a different function and is used differently. In the above example, the expression `x = True` represents a value, and thus the = sign is interpreted as a comparison operator. In a stand-alone statement, it would be interpreted as an assignment operator and assign the value on the right to the variable on the left. The following example illustrates this:

```
If x = True // Compare the value of x to True
  y = False // Assign False to variable y
EndIf
```

Comparison operators, such as =, <, >, <>, <=, and >=, produce Boolean expressions by comparing the expression on the left side of the operator to the expression on the right side of the operator and evaluating the result as true or false. For instance:

```
39 < 81 // Evaluates to True
```

Comparison expressions can be combined using logical operators to produce more complex Boolean expressions. The following example demonstrates the use of comparison operators in conjunction with a logical operator:

```
x > y And x < 5000
```

In the example, the value of the overall expression depends on the values of the expressions on each side of the *And operator. If both expressions are true, then the overall expression evaluates as True . If either expression is false, then the entire expression evaluates toFalse.

Parenthetical Expressions

You can use parentheses to control the evaluation of Boolean expressions. Expressions enclosed by parentheses evaluate first. For multiple levels of nesting, precedence is granted to the most deeply-nested expressions. Within parentheses, evaluation proceeds according to the rules of operator precedence. For more information, see **Operator Precedence in Visual RPG**.

Objects

When you create an application in Visual RPG, you constantly work with objects. You can use objects provided by Visual RPG — such as controls or forms. You can even create your own objects, and define additional properties and methods for them. Objects act like building blocks for programs, as they let you to write a piece of code once and reuse it again and again.

What is an Object?

An object is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object.

Objects let you declare variables and procedures once and then reuse them whenever needed. For example, if you want to add a spell checker to an application you could define all the variables and support functions to provide spelling check functionality to your application. However, if you create your spell checker as a class, you can reuse it in other applications by adding a reference to the compiled assembly. Better yet, you may be able to save yourself some work by using a spell checker class that someone else has already developed.

Each object in Visual RPG is defined by a class. Classes describe the fields, properties, methods, and events of an object. Objects are instances of classes; you can create as many objects you need once you have created a class.

To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie — for instance, size and shape. The class is used to create objects. The objects are the cookies.

Two examples of the relationship between classes and objects in Visual RPG may make this clearer:

- The controls on the Toolbox in Visual RPG represent classes. When you drag and drop a control from the toolbox onto a form, you are actually creating an object —an instance of a class.
- The form you work with at design-time is a class. At run-time, Visual RPG creates an instance of the form's class.

Objects are created as identical copies of their class. Once they exist as individual objects, however, their properties can be changed. For example, if you add three check boxes to a form, each check box button object is an instance of the CheckBox class. The individual CheckBox objects share a common set of characteristics and capabilities (properties, fields, methods, and events), defined by the class. However, each has its own name, can be separately enabled and disabled, can be placed in a different location on the form.

Relation of Objects to Each Other

Objects can be related to each other in several ways. One such relationship is hierarchical. Classes that are derived from more fundamental classes are said to have a hierarchical relationship. Class hierarchies are useful when describing items that are a subtype of a more general class. Derived classes inherit members from the class they are based on, allowing you to add complexity as you progress in a class hierarchy.

Another way that objects are related is the object object-container relationship. Container objects encapsulate other objects. For example, an instance of class with a property that stores an object is a container object.

One type of object containment is represented by collections. Collections are like arrays of objects that can be enumerated. Visual RPG supports a specific syntax in the form of `If..Else..EndIf` syntax blocks that allow you to iterate the items of a collection. Additionally, collections often allow you to use a method named `Item` property to retrieve elements by their index or by associating them with a unique string. Collections are easier to use than arrays because they allow you to add or remove items without using indexes. Because of their ease of use, collections are often used to store forms and controls.

The Basics of Working with Objects

Almost everything you do in Visual RPG involves objects. This topic shows you some ways that the elements of objects are used.

Member Access

You access the members of an object by specifying, in order, the name of the object, a period, and the name of the member you want to use. For example, if you have an object named Obj1 that has a property named Prop1 you could use the syntax `Obj1.Prop1` to access that property. In some cases you need to use several dots because objects can contain other objects. For example, you could use the syntax `Obj1.Prop1.X` if the property Prop1 is an object that contains a field named X.

Fields and Properties

Properties and fields represent information stored in an object. The values of fields and properties are retrieved and set with assignment statements the same way that variables are retrieved and set. The following example sets and retrieves the value of a field and a property.

```
DclFld X Type (*Integer) Len (4)
x = 8 DclFld
TheOtherForm Type( Form2 ) New( *Dft ) // Declares an instance of a class.
```

Methods

Methods are procedures defined within a class. Methods are used like subroutines or functions, but are accessed by way of the object to which they belong. The following code fragment calls a function and a subroutine declared in an object:

```
DclFld TheOtherForm Type( Form2 ) New( *Dft ) // Declares an instance of a class
DclFld Y Type (*Integer)
// Retrieves a value from a method declared as a function.
Y = Obj2.GetNumber
Obj2.StoreNumber(Y) // Calls a method declared as a subroutine.
```

Objects, Properties, Methods, and Events

Objects, properties, methods, and events are the basic units of object-oriented programming. An object is an element of an application, representing an *instance* of a class. Properties, methods, and events are the building blocks of objects, and constitute their *members*.

Objects

An object represents an instance of a class such as Form or Control. In Visual RPG code, you must **instantiate** an object before you can apply one of the object's methods or change the value of one of its properties. Instantiation is the process by which an instance of a class is created and assigned to an object variable. An example is shown below:

```
DclFld x Type (*Form) New (*Dft)
```

In this example, the variable `x` is assigned to refer to a new instance of the class `MyClass`.

Properties

A property is an attribute of an object that defines one of the object's characteristics, such as size, color, or screen location, or an aspect of its behavior, such as whether it is enabled or visible. To change the characteristics of an object, you change the values of its corresponding properties.

To set the value of a property, affix the reference to an object with a period, the property name, an equal sign (=), and the new property value. For example, the following code changes the caption of a Visual RPG Windows Form by setting the Text property:

```
BegSr ChangeCaptionName
  myForm.Text = myTitle
EndSr
```

Note that you can't set some properties. The Help topic for each property indicates whether you can set that property (read/write), only read the property (read-only), or only write the property (write-only).

You can retrieve information about an object by returning the value of one of its properties. The following procedure uses a message box to display the title that appears at the top of the currently active form.

```
BegSr GetFormName()
  DclFld formName Type(*String)
  formName = myForm.Text
  MsgBox(formName)
EndSr
```

Methods

A method is an action that an object can perform. For example, Add is a method of the ComboBox object, because it adds a new entry to a combo box.

The following example demonstrates the Start method of a Timer component:

```
// Instantiates a Timer object.
DclFld
myTimer Type (System.Windows.Forms.Timer) New (*Dft)
// Invokes the Start method of myTimer.
MyTimer.Start
```

Events

An event is an action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. Events can occur as a result of a user action or program code, or they can be triggered by the system. You can also develop your own custom events to be raised by your objects and handled by other objects.

Using Constructors

Constructors control the creation of objects.

To create a constructor for a class, create a procedure named BEGCONSTRUCTOR anywhere in the class definition. To create a parameterized constructor, specify the names and data types of arguments to DclFld *New just as you would specify arguments for any other procedure.

When you define a class derived from another class, the first line of a constructor must be a call to the constructor of the base class, unless the base class has an accessible constructor that takes no parameters. Use ENDCONSTRUCTOR to destroy and release resources.

Operation Codes

An *operation code*, or *command in Visual RPG* is a complete instruction that can contain parameters, operators, variables, constants, and expressions. All op codes fall into one of two categories: *declaration commands*, which name a variable, constant, or procedure and can also specify a data type; or *executable commands*, which initiate actions.

An operation code, or command in Visual RPG is a complete instruction. It can contain keywords, operators, variables, constants, and expressions. Each command belongs to one of the following two categories:

- Declaration commands, which name a variable, constant, or procedure and can also specify a data type. For more information, see Declaration Operation Codes.
- Executable commands, which initiate actions. These commands can execute a method or function, and they can loop or branch through blocks of code. Executable commands include assignment commands, which assign a value or expression to a variable or constant. For more information, see Executable Operation Codes and Assignment Operation Codes.

Continuing a Statement over Multiple Lines

An operation code usually fits on one line, but when it doesn't, you can continue a long command onto the next line using a line-continuation character, which consists of a plus sign. In the following example, the DCLFLD command is continued over two lines:

```
DCLFLD myVar Type (*String) Shared (*YES) +  
WithEvents (*YES)
```

Adding Comments

To help document code, it is useful to make liberal use of embedded comments. Comments in code can explain a procedure or a particular instruction to anyone reading or working with it later. Visual RPG ignores comments, however, when it runs your procedures.

Comment lines begin with a double slash (`//`), and can be added anywhere in code. Comments can also go on their own separate line. To extend comments over multiple lines, you can either add a double slash at the beginning of each line, or use a `*/` at the beginning of the comment and end the comment with a `/*`.

Assignment Operation Codes

Assignment operation code or commands carry out assignment operations. Simple assignment operations consist of taking the value on the right side of the operator and assigning it to the variable on the left, as in this example:

```
x = 42
```

Here, the variable `x` is assigned the literal value 42.

The value on the right side of the operator can be any literal, variable, expression, or function call that returns a value, as in this example:

```
x = y + z + MyFunction(3)
```

Here, the value represented by `y` is added to the value represented by `z`, which is added to the value returned by the call to `MyFunction(3)`. The total value of this expression is then assigned to the variable `x`.

The assignment operator can also assign values to String variables and String expressions, as in this example:

```
Name(x) Type (*CHAR) Len (10)  
x = "String variable assignment"  
x = "Con" & "cat" & "enation" // x equals "Concatenation".
```

The values of Boolean variables can be assigned as well, using either a Boolean literal or a Boolean expression as the right argument:

```
DCLFLD x Type (*Boolean)  
x = True  
x = 45 > 1003 // x equals *False.  
x = 45 > 1003 Or 45 > 17 // x equals *True.
```

The argument on the left side of the operator can be any valid variable or property. The following example demonstrates setting the value of the text property of a text box:

```
MyTextBox.Text = "This " & "is a " & "String"  
// MyTextBox.Text = "This is a String".
```

Other assignment operation code or commands first perform an operation on an argument before assigning it to another argument. The following example illustrates one of these operators, +=, which increments the value of the variable on the left side of the operator by the value of the expression on the right:

```
x += 1
```

In this example, 1 is added to the value of `x`, and that new value is then assigned to `x`. It is a shorthand equivalent of the following operation code or command:

```
x = x + 1
```

A variety of compound assignment operations can be performed using operators of this type. If the value of the variable on the left side of the operator has not been set, it is treated as zero. For a list of these operators and more information about them, see Assignment Operators in the Visual RPG Language Reference.

Lastly, the concatenation assignment operator is useful for adding a string to the end of already existing strings, as illustrated in this example:

```
DCLFLD x Type (*String) = ("My ")  
x &= ("String" ) x = ("My String").
```

If the variable on the left side of the concatenation assignment operator has not been assigned a value, it is treated as an empty string ("").

Declaration Operation Codes

Declaration operation codes are used to name and define subroutines, variables, arrays, and constants. When you declare a procedure, variable, or constant, you also define its access, depending upon where you place the declaration and what keywords you use to declare it.

A declaration operation code reserves the memory needed to create a variable, but does not explicitly create it. You can explicitly create a variable and assign it a value at the same time by using the following syntax:

```
// MyField is created and assigned the initial value 45.  
DCLFLD Name( MyField ) Type (*Integer) Len( 2) Inz( 45 )
```

If your variable is an object variable, you can explicitly create an instance of its class when you declare it by using the `New` keyword, as illustrated below:

```
DclFld TheOtherForm Type( Form2 ) New( *Dft )
```

The following is a listing of declaration operation codes and those that allow you to specify an access type.

- BEGCLASS
- BEGCONSTRUCTOR
- BEGENUM
- BEGFUNC
- BEGININTERFACE
- BEGPROP
- BEGSR
- DCLALIAS

- DCLARRAY
- DCLCONST
- DCLDB
- DCLDISKFILE
- DCLDS
- DCLEVENT
- DCLFLD
- DCLMEMORYFILE
- DCLSUBFILE
- DCLSUBFILEFLD
- DCLWORKSTNFILE

Executable Operation Codes

An executable command initiates action by executing a method. It can loop or branch through blocks of code. Executable commands often contain mathematical or conditional operators.

The following example uses an If...Then...Else command to execute different blocks of code based on the value of a variable.

```
BegSr StartCounter
  DCLFLD counter Type (*Integer)
  If clockwise =
*True
  Then .....
  .....
  Else
  .....
  .....
  .....
EndIf
EndSr
```

The If...Else command block always end with an EndIf.

Operators

An operator performs an operation on one or more value-returning code elements. Such an operation can be an arithmetic operation such as addition or multiplication; a concatenation operation that combines two strings into a new string; a comparison operation that determines which of two values is greater; a logical operation evaluating whether two expressions are both true. Operators combined with value-returning code elements such as literals or variables form expressions.

The following is an example of a statement that uses the simple assignment operator (=):

```
x = 85
```

In the preceding example, the assignment operator performs the assignment operation: it takes the value on the right side of the operator and assigns it to the variable on the left. Several operators can perform actions in a single expression or statement, as demonstrated below:

```
x = 85 + y * z ^ 2
```

In this example, the operations in the expression on the right side of the assignment operator are performed, and the resulting value is assigned to the variable `x`, on the left. There is no limit to the number of operators that can be combined into an expression, but an understanding of Operator Precedence is necessary to ensure that you get the results you expect.

Arithmetic operators

Arithmetic operators are used to perform many of the familiar arithmetic operations that involve the calculation of numeric values represented by literals, variables, other expressions, function and property calls, and constants.

Arithmetic Operations

You can add two values in an expression together with the `+` operator, or subtract one from another with the `-` operator, as shown in the following examples:

```
DCLFLD x Type (*Integer) Len (4)
```

```
x = 67 + 34
```

```
x = 32 - 12
```

Negation also uses the `-` operator, but with somewhat different syntax, as shown below:

```
DCLFLD x Type (*Integer) Len (4)
```

```
x = 65
```

```
DCLFLD y As Integer
```

```
y = -x
```

Multiplication and division use the `*` and `/` arithmetic operators, respectively, as in these examples:

```
DCLFLD y Type (*Integer) Len (4)
```

```
y = 45 * 55.23
```

```
y = 32 / 23
```

Exponentiation uses the `^` operator, as shown below:

```
DCLFLD z Type (*Integer) Len (4)
```

```
z = 23 ^ 3 // z equals the cube of 23.
```

Integer division is carried out using the `\` operator. Integer division returns the number of times one integer can evenly divide into another. Only integral types (`*Byte`, `*Float`, and `*Integer`) can be used with this operator. All other types must be converted to an integral type first. The following is an example of integer division:

```
DCLFLD k Type (*Integer)
```

```
k = 23 \ 5 // k = 4
```

Division by Zero

Division by zero has different results depending on the data types involved. In integral divisions (`*Integer`, `*Packed`, `*Float`), a `System.DivideByZeroException` is thrown. In division operations on the `*Decimal` or `*Packed` data type, Visual RPG also throws a `System.DivideByZeroException` exception. In floating-point divisions with the `Double` variable type, however, the result is the class member

representing NaN, PositiveInfinity, or NegativeInfinity, depending on the dividend. The following table summarizes the various results of attempting to divide by zero.

Bitwise Operations

Bitwise operations evaluating two values in binary (base 2) form, compare the bits at corresponding positions, and then assign values based on the comparison, as in the demonstration below of bitwise And:

```
DCLFLD x Type (*Integer) Len (4)
x = 3 And 5 // x = 1
```

In this example, the value of `x` is 1. Here's why:

- First, the values are converted to binary form, as shown below:
3 in binary form = 011
5 in binary form = 101
- Next, the binary representations are compared, one binary position at a time. If both bits at a given position are 1, then a 1 is placed in that position in the result. If either bit is 0, then a 0 is placed in that position in the result. In our example this works out as follows:

```
011   3 in binary form
101   5 in binary form
001   The result, in binary form
```

Converting back to decimal, 001 is the binary representation of 1, so `x = 1`.

The bitwise `*BitOr` operation is similar, except that a 1 is assigned to the result bit if either of the compared bits is 1. `*BitXor` assigns a 1 to the result bit if only one of, but not both of, the compared bits is 1. `*Not` takes a single operand and inverts all the bits, including the sign bit, and assigns that value to the result. This means that for positive numbers, `*Not` always returns a negative. Likewise, for negative numbers, `*Not` always returns a positive or zero.

Note Bitwise operations can be performed on integral types only. Floating point values must be converted to integral types before bitwise operation can proceed.

Bit Shift Operations

A bit shift operation performs an arithmetic shift on a bit pattern. The pattern is contained in the operand on the left, while the operand on the right specifies the number of positions to shift the pattern. The data type of the pattern operand must be `*Byte`, `*Integer`, or `*DECIMAL | *FLOAT`. The data type of the shift amount operand must be `*Integer` or must widen to `*Integer`.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. The bit positions vacated by a shift are set as follows:

- Zero for an arithmetic left shift
- Zero for an arithmetic right shift of a positive number
- Zero for an arithmetic right shift of any pattern with data type `*Byte`
- One for an arithmetic right shift of a negative number (`*Integer`)

In the following example, an Integer value is shifted both left and right:

```
DCLFLD Pattern Type (*Integer)Len (4)
x= 12 // Low-order bits are 0000 1100
LResult Type (*Packed) *As Type (*Integer)
```

```
LResult = Pattern << 3 // Left shift of 3 bits produces value of 96
RResult = Pattern >> 2 // Right shift of 2 bits produces value of 3
```

Arithmetic shifts never generate overflow exceptions.

Type Safety

Operands should normally be of the same type. For example, if you are adding an *Integer variable, you should add it to another *Integer variable, and the variable that the value is assigned to should be of type *Integer as well.

Comparison Operators

Comparison operators compare two expressions and return a *Boolean value that represents the result of the comparison. There are operators for comparing numeric values, operators for comparing strings, and operators for comparing objects. All three types of operators are discussed below.

Comparing Numeric Values

Numeric values are compared using six numeric comparison operators. The following table lists the operators and gives an example of the condition tested by the operator.

Operator	Condition tested	Examples
= (Equality)	Is the value represented by the first expression equal to the value represented by the second?	59 = 72 // False 59 = 59 // True 59 = 8 // False
<> (Inequality)	Is the value represented by the first expression not equal to the value represented by the second?	59 <> 72 // True 59 <> 59 // False 59 <> 8 // True
< (Less than)	Is the value represented by the first expression less than the value represented by the second?	59 < 72 // True 59 < 59 // False 59 < 8 // False
> (Greater than)	Is the value represented by the first expression greater than the value represented by the second?	59 > 72 // False 59 > 59 // False 59 > 8 // True
<= (Less than or equal to)	Is the value represented by the first expression less than or equal to the value represented by the second?	59 <= 72 // True 59 <= 59 // True 59 <= 8 // False
>= (Greater than or equal to)	Is the value represented by the first expression greater than or equal to the value represented by the second?	59 >= 72 // False 59 >= 59 // True 59 >= 8 // True

Comparing Strings

Strings can be compared using the Like operator as well as using the numeric comparison operators. The Like operator allows you to specify a pattern; the string is then compared against the pattern, and if it matches, the result is True. Otherwise, the result is False. The numeric operators allow you to compare String values based on their sort order, as in this example:

```
"73" < "9" // This result is True.
```

The result is True because the first character in the first string sorts before the first character in the second string. If the first characters were equal, the comparison would continue to the next character in both strings, and so on. Equality of strings can also be tested using the equality operator, as in this example:

```
"101" = "101" // This result is True.
```

In this example, the two strings are directly compared, and, because they are equal, True is returned.

If one string is a prefix of another, such as "ab" and "aba", the longer string sorts after the shorter. Thus the following expression is true:

```
"aba" > "ab"
```

The sort order will be based on either a binary comparison or a textual comparison depending on the Option Compare setting. For more information see Option Compare Statement.

Comparing Objects

In this example, `x Is y` evaluates to True, because both variables refer to the same instance. Contrast this result with the following example:

```
DclFld x New  
MyClass() DclFld y New MyClass()  
If x Is y then  
// Insert code to continue program execution.
```

In this example, `x Is y` evaluates to False, because although the variables are of the same type, they refer to different instances of that type.

You can test whether an object is of a particular type with the `*TypeOf...*Is` operator. . The syntax is

```
*TypeOf <object expression> *Is <TypeName>
```

When `TypeName` specifies an interface type then the `TypeOf ... Is` operator returns true if the object implements the interface type. When the `TypeName` is a class type then the operator returns true if the object is an instance of the specified class or is an instance of a class that derives from the specified class. For example:

```
DclFld x  
New (System.Windows.Forms.Form.Button) x = New Button()  
// Insert code to continue program execution.
```

Since the type of `x` is `Button`, and `Button` inherits from `Control`, the `*TypeOf x Is Control` expression evaluates to True.

Concatenation Operators

Concatenation operators join multiple strings into a single string. There are two concatenation operators, `+` and `&`; both carry out the basic concatenation operation, as shown below:

```
DCLFLD x Type (*String)  
x = "Con" & "caten" & "ation" // x equals "Concatenation".  
x = "Con" + "caten" + "ation" // x equals "Concatenation".
```

These operators can also concatenate String variables, as in the following example:

```
DCLFLD x Type (*String)= "abc"  
DCLFLD y Type (*String)  
x = "abc"  
y = "def"
```

```
DCLFLD zType (*String)
z = x & y // z equals "abcdef".
z = x + y // z equals "abcdef".
```

Logical Operators

The logical operators `*BitAnd` and `*BitOr` are used to perform a logical disjunction on two Boolean expressions, or a bitwise disjunction on two integer values. The `*BitAnd` and `*BitOr` operators take two operands. The `*BitNot` operator specifies the result of a bitwise NOT operation performed on a numeric value. The `*BitNot` operator takes a single operand.

The `*BitNot` operator specifies the result of a bitwise NOT operation performed on a numeric value. If the numeric value is not an integer, it is converted to an integer before its bits are shifted.

The `*BitAnd` operator performs a logical conjunction on two Boolean expressions, or a bitwise conjunction on two integer values. For Boolean comparisons, the result is the logical conjunction of two expressions. For bitwise operations, the result is a numeric value resulting from the bitwise conjunction of two numeric expressions.

The `*BitOr` operator performs a logical disjunction on two Boolean expressions, or a bitwise disjunction on two integer values. For Boolean comparisons, the result is the logical disjunction of two expressions. For bitwise operations the result is a numeric value resulting from the bitwise disjunction of two numeric expressions.

Strings

The `*String` data type represents a series of characters (each representing in turn an instance of the `*Char` data type). An instance of a string can be assigned to a literal value that represents a series of characters. For example:

```
DclFld MyString Type (*String)
MyString = "This is an example of the String data type"
```

A `*String` variable can also accept any expression that evaluates to a string. Examples are shown below:

```
DclFld String1 Type (*String)
DclFld String2 Type (*String)
String1 = "one, two, three, four, five"
String2 = OneString.Substring(5,3) // Evaluates to "two".
String1 = "1"
String2 = OneString & "1" // Evaluates to "11".
```

Any literal that is assigned to a `*String` variable must be enclosed by quotation marks ("). Naturally, this makes creating a literal that contains quotation marks somewhat problematic. For example, the following code returns a compiler error:

```
DclFld myString
Type (*String) myString = "Look at this example!" // This causes an error.
```

This code returns a compiler error because the compiler terminates the string after the second quotation mark, leaving the rest of the string hanging. To have a quotation mark represented in a string literal, you must represent it as two quotation marks. The following example demonstrates the correct way to include a quotation mark in a string:

```
DclFld myString Type (*String) // The value of myString is: "Look at
this example!" myString = ""Look at this example!""
```

In the above example, the two quotation marks preceding the word 'Look' exist in the string as a single quotation mark. The three quotation marks at the end of the line represent a quotation mark in the string and the string termination character.

A string can be thought of as a series of `*Char` instances, and the `*String` type has built-in functions that allow you perform many manipulations on a string that resemble the manipulations allowed by arrays.

The Immutability of Strings

A string is *immutable*, which means its value cannot be changed once it has been created. How, then, does the following example make any sense?

```
DclFId myString Type (*String) = "This string is immutable"  
myString = "Or is it?"
```

In the above example, it appears that a string is created, given a value, and then has that value changed. This appears to contradict the opening statement that strings are immutable. What is actually happening is that in the first line, an instance of String is created and given the value "This string is immutable". In the second line of the example, that *instance* is destroyed and a new instance is created and given the value "Or is it?". The only thing that doesn't change is the name of the variable.

Unlike other intrinsic data types, *String is a reference type. When a variable of reference type is passed as an argument to a function or subroutine, a reference to the memory address where the data is stored is passed instead of the actual value of the string. So in the previous example, the name of the variable remains the same, but it points to a new and different instance of the String class, which holds the new value.

Manipulating Strings

The String class of the .NET framework provides many built-in methods to facilitate the comparison and manipulation of strings. It is now a trivial matter to get data about a string, or to create new strings by manipulating current strings. The Visual RPG language also has inherent methods that duplicate many of these functionalities.

Types of String Manipulation Methods

In this section you will read about several different ways to analyze and manipulate your strings. Some of the methods are a part of the Visual RPG language, and others are inherent in the String class.

Visual RPG methods are used as inherent functions of the language. They may be used without qualification in your code. The following example shows typical use of a Visual RPG string-manipulation command:

```
DclFId aString Type(*String)  
Inz("SomeString") DclFId bString Type(*String)
```

You can also manipulate strings with the methods of the **String** class.

There are two types of methods in **String**: *shared* methods and *instance* methods.

A shared method is a method that stems from the String class itself and does not require an instance of that class to work. These methods can be qualified with the name of the class (String) rather than with an instance of the String class. For example:

```
DclFId aString Type (*String)  
bString = String.Copy("A literal string")
```

In the preceding example, the String.Copy method is a static method, which acts upon an expression it is given and assigns the resulting value to `bString`.

Instance methods, by contrast, stem from a particular instance of String and must be qualified with the instance name. For example:

```
DclFId aString Type(*String) Inz("A  
String") DclFId bString Type(*String)  
bString = aString.SubString(2,6) // bString = "String"
```

In this example, the SubString method is a method of the instance of String (that is, `aString`). It performs an operation on `aString` and assigns that value to `bString`.

Nothing and Strings

The Visual RPG runtime and the .NET Framework evaluate *Nothing differently when it comes to strings. Consider the following example:

```
DclFId MyString Type (*String) Inz("This is my string")
DclFId stringLength As Integer
' Explicitly set the string to Nothing.
MyString = Nothing
' stringLength = 0
stringLength = Len(MyString)
' This line, however, causes an exception to be thrown.
stringLength = MyString.Length
```

An un-instanced string is evaluated as Nothing, and an empty string is evaluated as *Blanks or "".

Comparing Strings

You can compare two strings by using the String.Compare method. This is a static, overloaded method of the base string class. In its most common form, this method can be used to directly compare two strings based on their alphabetical sort order. The following example illustrates how this method is used:

```
DclFId myString Type(*String)
Inz("Alphabetical") DclFId secondString Type *String) Inz("Order")
DclFId result As Integer
result = String.Compare (myString, secondString)
```

This method returns an integer that indicates the relationship between the two compared strings based on the sorting order. A positive value for the result indicates that the first string is greater than the second string. A negative result indicates the first string is smaller, and zero indicates equality between the strings. Any string, including an empty string, evaluates to greater than a null reference.

Additional overloads of the String.Compare method allow you to indicate whether or not to take case or culture formatting into account, and to compare substrings within the supplied strings. For more information on how to compare strings, see String.Compare Method.

Searching for Strings Within Your Strings

There are times when it is useful to have data about the characters in your string and the positions of those characters within your string. A string can be thought of as an array of characters (Char instances); you can retrieve a particular character by referencing the index of that character through the Chars property. For example:

```
DclFId myString Type(*String)
Inz("ABCDE") DclFId myChar Type(*Char)
myChar = myString.Chars(3) // myChar = "D"
```

You can use the String.IndexOf method to return the index where a particular character is encountered, as in the following example:

```
DclFId myString Type(*String)
myString =
"ABCDE" DclFId myInteger Type(*Integer)
myInteger = myString.IndexOf("D") // myInteger = 3
```

In the previous example, the IndexOf method of `myString` was used to return the index corresponding to the first instance of the character "C" in the string. IndexOf is an overloaded method, and the other overloads provide methods to search for any of a set of characters, or to search for a string within your string, among others. The Visual RPG command InStr also allows you to perform similar functions.

Creating New Strings from Old

When using strings, you may want to modify your strings and create new ones. You may want to do something as simple as convert the entire string to uppercase, or trim off trailing spaces; or you may want to do something more complex, such as extracting a substring from your string. The `System.String` class provides a wide range of options for modifying, manipulating, and making new strings out of your old ones.

To combine multiple strings, you can use the concatenation operators (& or +). You can also use the `String.Concat` Method to concatenate a series of strings or strings contained in objects. An example of the `String.Concat` method follows:

```
DclFld aString Type(*String)
Inz("A") DclFld bString Type(*String) Inz("B") DclFld
cString Type(*String) Inz("C") DclFld dString Type(*String)
Inz("D") DclFld myString Type(*String)
' myString = "ABCD"
myString = String.Concat(aString, bString, cString, dString)
```

You can convert your strings to all uppercase or all lowercase strings using either the Visual RPG functions `UCase` Function and `LCase` Function or the `String.ToUpper` Method and `String.ToLower` Method methods. An example is shown below:

```
DclFld myString Type(*String) Inz("UpPeR oR LoWeR
cAsE") DclFld newString Type(*String)
// newString = "UPPER OR LOWER CASE"
newString = UCase(myString)
// newString = "upper or lower case"
newString = LCase(myString)
// newString = "UPPER OR LOWER CASE"
newString = myString.ToUpper
// newString = "upper or lower case"
newString = myString.ToLower
```

The `String.Format` method and the Visual RPG `Format` command can generate a new string by applying formatting to a given string. For information on these commands, see `Format` Function or `String.Format` Method.

The `String.Trim` functions and related functions also allow you to remove instances of a specific character from the ends of your string. The following example trims all leading and trailing instances of the "#" character:

```
DclFld myString Type(*String) Inz("#####Remove
those!#####") DclFld oneString Type(*String)
OneString = myString.Trim("#")
```

You can also add leading or trailing characters by using the `String.PadLeft` Method or the `String.PadRight` Method.

If you have excess characters within the body of your string, you can excise them by using the `String.Remove` Method, or you can replace them with another character using the `String.Replace` Method. For example:

```
DclFld aString Type(*String) Inz("This is
My Str@@@o@@@ing") DclFld myString
Type(*String) DclFld anotherString Type(*String)
// myString = "This is My String"
myString = aString.Remove(14, 5)
```

```
// anotherString = "This is Another String"
anotherString = myString.Replace("my", "Another")
```

You can use the `String.Replace` method to replace either individual characters or strings of characters. The Visual RPG Mid Statement can also be used to replace an interior string with another string.

You can also use the `String.Insert` Method to insert a string within another string, as in the following example:

```
DclFld aString Type(*String) Inz("This is My
String") DclFld myString Type(*String)
// Results in a value of "This is My String".
myString = aString.Insert(13, "ri")
```

The first parameter that the `String.Insert` method takes is the index of the character the string is to be inserted after, and the second parameter is the string to be inserted.

You can concatenate an array of strings together with a separator string by using the `String.Join` Method. Here is an example:

```
DclFld shoppingItem(2)
Type(*String) DclFld shoppingList Type(*String)
shoppingItem(0) = "milk"
shoppingItem(1) = "Eggs"
shoppingItem(2) = "Bread"
shoppingList = String.Join(",", shoppingItem)
```

The value of `shoppingList` after running this code is "milk,Eggs,Bread". Note that if your array has empty members, the method still adds a separator string between all the empty instances in your array.

You can also create an array of string from a single string by using the `String.Split` Method. The following example demonstrates the reverse of the previous example: it takes a shopping list and turns it into an array of shopping items. The separator in this case is an array of characters.

```
DclFld shoppingList Type (*String) Inz("milk,Eggs,Bread")
DclArray shoppingItem Type( *String ) Rank( 1 )
shoppingItem = shoppingList.Split( ",", ToCharArray() )
```

Substrings of your string can also be generated using the `String.Substring` Method. This method takes two arguments: the character index where the substring is to start, and the length of the substring. The `String.Substring` method operates much like the `Mid` function. An example is shown below:

```
DclFld aString Type (*String)
aString = "Left Center Right"
DclFld subString Type (*String)
// subString = "Center"
subString = aString.SubString(5,6)
```

Variables

Variable Declaration

You declare a variable to specify its name and characteristics. The declaration statement for variable declaration is the `DCLFLD` statement. Its location and contents determine the variable's characteristics.

Declaration Levels

A *local variable* is one that is declared within a procedure. A *module variable* is declared at module level, inside the module but not within any procedure internal to that module.

In a class or structure, the category of a nonlocal variable depends on whether or not it is shared. If it is declared with the Shared keyword, it is a *shared variable*, and it exists in a single copy shared among all instances of the class or structure. Otherwise it is an *instance variable*, and a separate copy of it is created for each instance of the class or structure. A given copy of an instance variable is available only to the instance for which it was created.

Declaring Data Type

The *As data type in the declaration statement allows you to define the data type or object type of the variable you are declaring. You can specify any of the following types for a variable:

- An elementary data type, such as *Boolean, or *Integer.
- A composite data type, such as an array or structure.
- An object type, or class, from Visual RPG or another application, such as Label or TextBox.

In the following statement, the variable declaration `x` is declared as type *Integer and changed to *Boolean:

```
DCLFLD x Type(*Integer) *As (*Boolean).
```

For more information on data types, see **Data Types**.

Declaring Lifetime

The *lifetime* of a variable is the period of time during which it is available for use. A local variable declared with a DCLFLD statement exists only as long as its procedure is executing. When the procedure terminates, all its local variable declaration disappear and their values are lost. However, if you declare a local variable using the Static (*YES) keyword, it continues to exist and preserve its value even when the procedure ends.

For more information on lifetime, see **Lifetime**.

Declaring Accessibility

The *accessibility, or scope* of a variable is the set of all code that can refer to it without qualifying its name. A variable's scope is determined by where the variable is declared. Code located in a given region can use the variable declaration defined in that region without having to qualify their names. When declaring scope, the following rules apply:

- The scope of a module variable is the entire namespace in which the module is defined.
- The scope of a shared or instance variable is the structure or class in which it is declared.
- The scope of a local variable is the procedure in which it is declared.
- However, if you declare a local variable within a block, its scope is that block only. A block is a set of statements terminated by an If...Then...Else...End If construction.

Object Variables

In addition to storing values, a variable can refer to an object. You assign an object to a variable for the same reasons you assign any value to a variable:

- A variable name is often shorter and easier to remember than the full path of methods or properties necessary to access the object itself.
- Using a variable that refers to an object is more efficient than repeatedly accessing the object itself through the necessary methods or properties.

- You can change a variable to refer to other objects while your code is running.

Object Variable Declaration

You use a normal declaration statement to declare an object variable, but for the data type, you specify either `Object` or the *object class*. The object class is the specific class from which the object is to be instantiated. When you declare a variable with a specific object class, it can access all the methods and properties exposed by that class. If you declare the variable with `Object`, it can use only the members of the `Object` class.

Use the following syntax to declare an object variable:

```
DCLFLD variablename *New {objectclass | Object}
```

You can also use `*Protected`, `*Internal`, `*Private`, `*Public` for the declaration.

Sometimes the object class is not known until the procedure runs. In this case, you must declare the object variable with the `*Object` data type. .

Declaring an object variable as a specific object class gives you several advantages:

- Automatic type checking.
- Improved readability.
- Fewer errors in your code.
- Faster code execution.

Flexibility of Object Variables

When working with objects in an inheritance hierarchy, you have a choice of which class to use for declaring your object variables. In making this choice, you must balance flexibility of object assignment against access to a class's members. For example, consider the inheritance hierarchy that leads up to the `Form` class.

Suppose your application defines a form called `Form2`, which inherits from class `Form`. You can declare an object variable that refers specifically to `Form2`, as follows:

```
DclFld MyForm Type( *Form ) *New( Form2 ) //Can assign only Form2 objects to MyForm.
```

This limits the variable `MyForm` to objects of class `Form2`, but it also makes all the methods and properties of `Form2` available to `MyForm`, as well as all the members of all the classes from which `Form2` inherits.

You can make an object variable more general by declaring it to be of type `Form`, as follows:

```
DclFld AnyForm Type( *Form ) //Can assign any form, but no other control.
```

This allows you to assign any form in your application to `AnyForm`.

Object Variable Assignment

You use a normal assignment statement to assign an object to an object variable. You can assign an object expression or the `*Nothing` figurative constant.

`*Nothing` means there is no object currently assigned to the variable. When your code begins running, your object variables are initialized to `*Nothing`, except for those whose declarations include assignment.

You can combine assignment with declaration by using the `*New` operator.

Setting an object variable to `* Nothing` discontinues the association of the variable with any specific object. This prevents you from accidentally changing the object by changing the variable. It also allows you to test whether the object variable points to a valid object.

Current Instance of an Object

The *current instance* of an object is the one in which the code is currently executing. Since all code executes inside a procedure, the current instance is the one in which the procedure was invoked.

The *This figurative constant acts as an object variable referring to the current instance. If a procedure is not shared, it can use the *This constant to obtain a pointer to the current instance. Shared procedures cannot be associated with a single instance of an object.

Using *This is particularly useful for passing the current instance to a procedure in another module.