

#1) What is an object, anyway?

An object is a predefined set of data (variables), neatly packaged with a group of subroutines (code) which manipulate the data and provide any other functionality you need.

For example, a string array containing names and addresses (data) might be packaged with a subroutine (code) that displays a popup dialog to edit the data, another subroutine (code) to print mailing labels, and so forth. That's a great candidate for an object.

In short, an object is a complete little programming package, code and data, all in one tightly contained place. It's safer and protected, easier to debug, maintain, and reuse. An object can be written to perform most any task you might imagine.

In object terminology, a CLASS is used to define an object. A CLASS is much like an enhanced user-defined type; it's a description of both the variables and the subroutines which make up the object. When you create an object, the CLASS definitions are used to do so. Memory is allocated for the variables, pointers are directed to the subroutines, and all this information is made available to a program.

Each time an OBJECT is created, it's called an INSTANCE of the definition (an instance of the CLASS). That's why these variables are called INSTANCE variables. When you create multiple objects (from the same CLASS definition), each instance gets its own individual set of these INSTANCE variables, and each instance gets individual access to the subroutines.

Some programming languages use objects for everything. That's wrong. Better programming languages make them optional. Objects are a great programming tool for many things, but not everything. Standard Subs and Functions must always be supported, so you can blend the techniques to suit the programmer, not the compiler.

Thousands of books have been written to describe objects and object oriented programming. In most cases, the buzz words and abstract definitions make it seem as though they're designed to confuse, not enlighten. The world needs a good and straightforward explanation of objects in general, and COM objects in particular. We'll see if we can help fill that need.

A key trait of objects in general is the concept of encapsulation. Data is "hidden" within the object, so INSTANCE variables cannot be accessed from the outside.

INSTANCE variable data may only be set, altered, or retrieved by the subroutines in the object. These variables are hidden from the rest of the program.

Over the years, objects have gained a reputation for slow, bloated programming. In many cases, the reputation was well deserved. But don't let that fool you. Objects can execute just as fast as standard Subs and Functions. Objects can be just as tiny, and just as efficient as standard Subs and Functions. It just takes a little effort and a little dedication to the important goals of object programming.

#2) Where are objects located?

Since an object is a complete programming package (sort of like the idea of a sub-program), it can be located in many different places. However, regardless of where the object is found, a great compiler will still handle all the messy details for you... automatically.

In many cases, objects will be located right within your main program. You can create a single, self-contained program, with one object or a thousand objects. Get all the power of objects, but keep the details private -- for your eyes only.

Objects can be located in a Dynamic Link Library (DLL). This is usually called a COM object, but is also known as an OCX or an ActiveX object. The actual file extension is largely irrelevant. The subroutines offered by these objects are generally available to any program which knows the subroutine definitions, and wishes to access them. This type of object is known as an "in-process" object because it is loaded into the address space of the calling application, just like a standard DLL.

Objects can also be located in an executable program (EXE). In this case, the calling application is frequently called a "controller", as it can control how the executable operates by manipulating its objects. A good example of this functionality is Microsoft Word -- by simply calling object subroutines, you can load a DOC file, display it to the user, make changes, then save the new document. All under the control of your calling application. Once again, the object subroutines are generally available to any program which knows the subroutine definitions. This type of object is known as an "out-of-process" object because it does not share address space with the calling application.

Whenever an object is accessed outside of your program, the COM (Component Object Model) services of Windows are generally used to make the "connection" for you. COM is an important tool for every programmer. It will open many opportunities for you. But more about COM later...

#3) Why should I use objects?

Obviously, we've all written perfectly good and usable code without an object in sight. You can continue down that path, or you can choose to take advantage of COM, objects, and more...

* Objects help you maintain your code. Objects break up your project into small, easily viewed parts. Usually, the input and output is clearly defined. You have all of the code and all of the data right at your fingertips.

* Objects help you write bug-free code. When you keep an object small and well-defined, you greatly enhance the stability of your programs. Consider the comparison to procedural programming: With standard Subs and Functions, it's typical to create the data (variables) in the calling code, but manipulate the data in the target procedures when they are executed. This separation of code and data has caused some of the most insidious bugs known to programmers. When you need to extend the range of data to a larger data type, it's easy to change the code. A piece of cake, so to speak. But what about the data? Now you must search out every reference to every involved Sub and Function. Find every data creation, every data change, and every other reference to these variables. What are the chances of missing a critical one? Far too great to ignore.

* Objects help you re-use your code. Since the object contains all the subroutines, and all the data, how could it be easier? Put one object source in one include file... Or put it in one DLL... Just use it when you need it!

* Objects help with team programming. Objects are self-contained. All of the subroutines and all of the data, all in one concise place. It's easy to create a precise definition for each object, and there's little dependency between the implementation of various objects. Each team member builds an object, one at a time, so it all comes together neatly in the end.

* Objects are an increasingly popular standard. Do you need to access the Windows API? Many of the newer API functions (DirectX graphics, for example) use only an object interface, and nothing else. If you don't use objects, you simply can't access them. Do you want to control an important application, like an Internet browser, word processor, or spreadsheet? COM objects are the only way to do it. As time goes by, objects will only become more embedded in day-to-day programming. Don't be left behind.

#4) What are the parts of an object?

* **METHOD:** A subroutine, very similar to a user-defined Sub/Function. A method has the special attribute that it can access the variables stored in the object. A method can return a value like a Function, or return nothing, like a Sub.

* **PROPERTY:** This is a METHOD, but in a specific form, for a specific purpose. A PROPERTY has all the attributes of a standard METHOD. It has a special syntax, and is specifically used to read or write private data to/from the internal variables in an object. This helps to maintain the principle of "encapsulation". Properties are usually created in pairs, a GET PROPERTY to read a variable, and a SET PROPERTY to write to a variable. Paired properties use the same name for both, since the correct one will be chosen for you based upon the usage in your source code. You should note this important fact: Since a PROPERTY is a form of METHOD, all of the documentation about METHODS also applies to PROPERTIES, unless specifically stated otherwise.

* **INTERFACE:** A definition of a set of methods and properties which are implemented on an object. You might think of it as a list of DECLARE statements where the sequence of the Declares must be preserved. Remember, the interface is just the definition, not the actual code. Every interface is associated with a GUID (a 128-bit number or string) which uniquely identifies this particular interface from all other interfaces, anywhere in the world. This identifier is called the Interface ID, or IID for short.

An interesting note is that one particular interface definition may become a part of several different classes and objects. In fact, the internal code for an interface in CLASS A may be entirely different from the internal code for the same interface in CLASS B. Method names, parameters, and return values must be identical, but the internal code could vary significantly. An important point: interfaces are immutable. Once an interface has been defined and published, the Method and Property definitions (sequence, names, parameters, return values, etc.) may never be altered. If you find you must change or extend an interface, you would usually define a new interface instead.

* **CLASS:** A definition of a complete object, which may include one or more interfaces. This is the place where you declare INSTANCE variables, and write your code for the enclosed METHOD and PROPERTY procedures. While some object implementations allow only a single interface per class, the better forms of objects (and COM objects in general) support the idea of optional multiple interfaces. Still, remember that a CLASS is the complete definition of an object. It defines all of the code and all of the data which will be found in a particular object. For this reason, there is only one copy of a CLASS. Every class is associated with a GUID (a 128-bit number or string) which uniquely identifies this particular class from all others, anywhere in the world. This identifier is called the Class ID, or CLSID. A friendlier version of the CLSID is a shorter text name, which also identifies the Class. This text name is known as the Program ID (PROGID), though it's possible this PROGID may not be totally unique. As it's a simpler construct, it might be duplicated in another program.

* **CLASS METHOD:** This is a private method, which may only be called from within the same CLASS. It is not a part of any interface, so it is never listed there. It is called a CLASS METHOD because it is a member of the class, not an interface. It is not visible to any code outside the class where it is defined. Code in a CLASS METHOD may call other CLASS METHODS in the same CLASS. Class Properties do not exist because there is no need for them. Within the object, variables can be accessed directly, so there is no need to use a PROPERTY procedure as an intermediary.

* **CONSTRUCTOR:** This is a special form of CLASS METHOD, which is executed automatically whenever an object is created. It is optional.

* **DESTRUCTOR:** This is a special form of CLASS METHOD, which is executed automatically whenever an object is destroyed. It is optional.

* **OBJECT:** An instance of a class. When you create an object in your running program, a block of memory is allocated for the set of instance variables you defined, and a virtual function table is established (a set of function code pointers) for each of the interfaces. You can create any number of OBJECTS based upon one CLASS definition.

It might be useful to think of an OBJECT in terms of an electrical appliance, like a television set. The TV is the equivalent of an OBJECT, because it's an instance of the plans which define all the things which make it a television. Of course, those plans are the equivalent of a CLASS. You can make many instances of a television from one set of plans, just as you can make many OBJECTS from one CLASS. The individual buttons and controls on the television are the equivalent of METHODS, while all of the controls, taken as a whole, are equivalent to the INTERFACE.

We don't need to know how a television works internally to use it and benefit from it. Likewise, we don't need to know how an object works internally to use it and benefit from it. We only need to know the intended job of the object, and how to communicate with it from the outside. The internal workings are well "hidden", which is called encapsulation. Since we can't "look inside" an Object, it's not possible to directly manipulate internal variables or memory. This provides a increased level of security for the internal code and data.

* **INSTANCE DATA:** Each CLASS defines some INSTANCE variables which are present in every object. When you create multiple objects (of the same class), each object gets its own unique copy of them. These variables are called INSTANCE variables because a new set of them is created for each instance of the object. For example, if you created a CUSTOMER object for each customer of your business, you might have INSTANCE variables for the Name, Address, Balance owed, etc. Each object would have its own set of INSTANCE variables to describe the attributes of that particular customer. INSTANCE variables are always private to the object. They can be accessed directly from any METHOD on the object, but they are invisible to any code outside of the object.

* **VIRTUAL FUNCTION TABLE:** Commonly called a VFT or VTBL, this is a set of function code pointers, one for each METHOD or PROPERTY in an interface. This is a tool used internally to direct program execution to the correct method or property you wish to execute. While it is a vital and integral part of every object, you need give it no concern other than to be aware of its existence. These items are managed for you, with no programmer intervention required.

#5) Are there other important "Buzz-Words"?

* **GUID:** This is a "Globally Unique Identifier", a very large number which is used to uniquely identify every interface, every class, and every COM application or library which exists anywhere in the world. GUID's identify the specific components, wherever and whenever they may be used. A GUID is a 16-byte (128-bit) value, which could be represented as an integer or a string. This 128-bit item is large enough to represent all the possible values, anywhere in the world. The PowerBASIC GUID\$() function (or a hot-key in the PowerBASIC IDE) can generate a random GUID which is statistically guaranteed to be unique from any other generated GUID. Each of these identifying GUID's may be assigned by the programmer, or they will be randomly assigned by the PowerBASIC compiler. When a GUID is written in text, it takes the form: {00CC0098-0000-0000-0000-0000000000FF}.

* **DIRECT INTERFACE:** This is the most efficient form of interface. When you call a particular METHOD or PROPERTY, the compiler simply performs an indirect jump to the correct entry point listed in the virtual function table (VFT or VTBL). This is just as fast as calling a standard Sub or Function, and is generally considered to be the best access method.

* **DISPATCH INTERFACE:** This is a slow form of interface, originally introduced as a part of Microsoft Visual Basic. When you use DISPATCH, the compiler actually passes the name of the METHOD you wish to execute as a text string. The parameters can also be passed in the same way. The object must then look up the names, and decide which METHOD to execute, and which parameters to use, based upon the text strings provided. This is a very slow process. Many scripting languages use DISPATCH as their sole method of operation, so continued support is necessary.

* **DUAL INTERFACE:** This is a combination of a Direct Interface and a Dispatch Interface. This most flexible form allows either option to be used, depending upon how the calling application is implemented.

* **AUTOMATION:** This is a special calling convention, defined by MS later in the evolution of COM and objects. An Automation object is simply one which adheres to the rules for Automation COM Objects. It may offer just a direct interface, just a

Dispatch interface, or both of them (DUAL). It should be noted that some programmers use the word AUTOMATION to mean DISPATCH. Even though that's not correct, you should keep the possibility in mind whenever you encounter the term. Automation Methods must use parameters, return values, and assignment variables which are AUTOMATION compatible: BYTE, WORD, DWORD, INTEGER, LONG, QUAD, SINGLE, DOUBLE, CURRENCY, OBJECT, STRING, and VARIANT. All Automation Methods return a hidden result code which is called the HRESULT. This is not really a handle, as the name suggests, but a result code to report the success or failure of a call to a METHOD or PROPERTY.

* IUNKNOWN: This is the name of a special interface which is the basis for every object. It has three methods, which are always defined as the first three methods in every interface. These 3 methods are used by compilers to look up other interfaces on the object, and to keep track of usage of this object. While IUNKNOWN is mandatory for every object, you won't ever need to reference it directly.

* OBJECT REFERENCE: This is a reference (a pointer) to an object, which is the only way objects are used. An object variable initially contains NOTHING. When you create an object, or duplicate one, a reference to that object is placed in an object variable by the compiler. That is, a pointer to the object is automatically inserted in the object variable. It is now considered to contain an OBJECT REFERENCE until such time as the reference is deleted or set to NOTHING.

* COMPONENT: An object that encapsulates code and data, providing a set of publicly available services.

* MONIKER: An object that implements the IMoniker interface. A moniker acts as a name that uniquely identifies a COM object. In the same way that a path identifies a file in the file system, a moniker identifies a COM object in the directory namespace.

#6) What should a Class look like?

In my opinion, the source code for a very simple class should look something like this. With just one interface and one instance variable, you'd see:

Code:

```
CLASS MyClass
  INSTANCE Counter as Long
  INTERFACE MyInterface
    INHERIT iUnknown          ' inherit the base class
    Method BumpIt(Inc as Long) as Long
      Temp& = Counter + Inc
      Incr Counter
      Method = Temp&
    End Method
  END INTERFACE
  ' more interfaces could be implemented here
END CLASS
```

Just like other blocks of code, a class would be enclosed in a CLASS statement and an END CLASS statement. Every class would be given a text name (in this case "MyClass") so it can be referenced easily in the program.

The INSTANCE statement would describe INSTANCE variables for this class. Each

object created from this class definition would contain its own private set of any INSTANCE variables. So, if you had a SHIRT class, you might have an instance variable named COLOR, among others. Then, if you create two objects from the class, the COLOR instance variable in the first object might contain WHITE, while the second might be BLUE.

Next would come the INTERFACE and END INTERFACE statements. They would define the one interface in this class, and they would enclose the methods and properties in this interface. Every interface would be given a text name (in this case "MyInterface") so it can be referenced easily in the program. You could add any number of additional interfaces to this class if it suited the needs of your program.

The first statement in every Interface Block would be the INHERIT statement. As mentioned earlier, every interface must contain the three methods of IUNKNOWN as its first three methods. In this case, INHERIT would be a shortcut, so you wouldn't have to type the complete definitions of those methods in every interface. There would be more complex (and powerful) ways to use INHERIT as well, but more about that later.

Finally, we have the METHOD and END METHOD statements. They are just about identical to a FUNCTION block, but they could only appear in an interface. In this case, the METHOD is named "BumpIt". It takes one ByRef parameter, and returns a long integer result.

How would you reference this object?

Code:

```
Function PBMain()  
  Dim Stuff as MyInterface  
  Let Stuff = Class "MyClass"  
  x& = Stuff.BumpIt(77)  
End Function
```

The first line of PBMain (DIM...) would define an object variable for an interface of the type "MyInterface". The LET statement would create an object of the CLASS "MyClass", and assigns an object reference to the object variable named "Stuff". The next line would execute the method "BumpIt", and assign the result to the variable "x&". It's just that simple!

#7) What is a Base Class?

The term "Base Class" is truly a misnomer, since it's actually an interface. The truth is, this term probably originated from those who use a programming language which supports only one interface per class. (Note: A much better plan is to allow an unlimited number of interfaces.) On those limited platforms, the distinction between a class and an interface tends to blur. However, since the term "Base Class" enjoys fairly wide usage already, it's probably best if we just learn to live with it and love it.

In our plan, every interface must ultimately derive from the IUnknown interface, since it provides information about an object that the compiler must have to manage these affairs accurately. Previously, we discussed the concept of adding INHERIT IUNKNOWN as the first line of every Interface Block. In that way, the appropriate code can be inserted automatically for you. so that the new interface you are

creating will derive all the functionality of IUNKNOWN, but still save you from all of that typing. What we didn't tell you at first was that there are really 3 System Base Classes available. The other two can be used, because they, too, are derived from IUNKNOWN.

So, the real definition of a Base Class is "The interface from which a newly created interface is derived". To implement any of the system interfaces, you would just use INHERIT followed by the Base Class name as the first line of the interface block. They are:

INHERIT IUNKNOWN

If this option is chosen, your methods may only be accessed using a Direct Interface, the most efficient form of access. It would use the STDCALL calling conventions, and return value conventions normally associated with C++. This style of Base Class is also known as a CUSTOM INTERFACE, so you can use "INHERIT CUSTOM" in place of "INHERIT IUNKNOWN" if that's more comfortable for you.

INHERIT IAUTOMATION

If this option is chosen, your methods may only be accessed using a Direct Interface, again the most efficient form of access. It uses the STDCALL calling conventions, and uses return value conventions involving a hidden parameter on the stack. Automation Methods must use parameters, return values, and assignment variables which are AUTOMATION compatible: BYTE, WORD, DWORD, INTEGER, LONG, QUAD, SINGLE, DOUBLE, CURRENCY, OBJECT, STRING, and VARIANT. All Automation Methods return a hidden result code which is called the HRESULT. This is not really a handle, as the name suggests, but a result code to report the success or failure of a call to a METHOD or PROPERTY. "AUTOMATION" is a synonym for "IAUTOMATION", so you can substitute "INHERIT AUTOMATION" in your code if that's more comfortable for you. Automation Interfaces have become more popular than Custom Interfaces in recent times, likely due to availability of the HRESULT hidden result code.

INHERIT IDISPATCH

If this option is chosen, you would automatically get a DUAL interface. That means your methods can be accessed using a Direct Interface (using Automation conventions described above), or the slower DISPATCH Interface, if that's what is needed. This is certainly the most flexible Base Class, and the only one which should be used if your methods will be accessed by code from a programming language other than PowerBASIC. In a DUAL interface, both forms return the HRESULT hidden result to report the success or failure of the operation. You may use the term "INHERIT DUAL" in place of "INHERIT IDISPATCH", if that's more comfortable for you. While a class may have any number of direct interfaces, only one DUAL or IDISPATCH interface is supported by the COM standard.

#8) What should an Interface look like?

An INTERFACE is a definition of a set of methods and properties which may be implemented on an object. Think of it as as much like a TYPE declaration, except that it contains Method and Property declarations instead of member variables. One interface definition may be used in many different classes and objects.

An Interface may appear in two general forms: the declaration form and the implementation form.

In the declaration form, the Interface just provides the "signature" of the member methods, without any other source code:

Code:

```
INTERFACE MyInterface
  INHERIT IAutomation
  METHOD Method1(parm AS LONG)
  PROPERTY GET Prop1() AS STRING
  PROPERTY SET Prop1(ByVal Text AS STRING)
END INTERFACE
```

This type of declaration interface can be used to provide a description of external interfaces, which you plan to access through COM services, or just as additional self-documentation of internal code.

In the implementation form, it is part of a CLASS definition, so it contains the complete source code to implement each of the member Methods and Properties.

Code:

```
CLASS AnyClass
  INTERFACE AnyInterface
    INHERIT IAutomation
    METHOD Method1(parm AS LONG)
      CALL abc(parm)
    END METHOD

    METHOD Method2(parm AS LONG)
      CALL abc(Parm*1)
    END METHOD
  END INTERFACE
END CLASS
```

In this case, you have the complete definition of an object, with code implemented so the methods can be called and executed.

The first entry in every INTERFACE block must be the base class upon which it is built. With this logic, you might choose one of the System Base Classes (IUnknown, IAutomation, or IDispatch), or you might decide to inherit a User Base Class instead.

Code:

```
INTERFACE CustomIface
  INHERIT IUnknown
  METHOD MethodDef()...
END INTERFACE
```

The above code defines a custom interface whose methods are available for direct access only. It uses custom calling conventions and does not support an HRESULT (OBJRESULT) return value.

Code:

```
INTERFACE AutoIface
  INHERIT IAutomation
  METHOD MethodDef()...
END INTERFACE
```

The above code defines an automation interface whose methods are available for direct access only. It uses automation calling conventions and always supports an HRESULT (OBJRESULT) return value. The above two forms will typically be used for internal objects, since they offer the best performance. Every internal interface and

every COM interface must ultimately inherit from IUnknown. As required base classes, the IUnknown and IAutomation declarations would have to be built into the compiler.

Code:

```
INTERFACE DispatchIface
  INHERIT IDispatch
  METHOD MethodDef()...
END INTERFACE
```

The above code defines a dual interface whose methods are available for both direct access and Dispatch access. This is the form you might typically use for COM objects, since it offers the best compatibility with varied client modules.

Every method and property in a dual interface needs a positive, long integer value to identify it. That integer value is known as a DispID (Dispatch ID), and it's used internally by COM services to call the correct function on a Dispatch interface. You can specify a particular DispID by enclosing it in angle brackets immediately following the Method/Property name in an Interface definition block.

Code:

```
INTERFACE DualIface
  INHERIT IDispatch
  METHOD MethodOne <76> ()
  METHOD MethodTwo <77> ()
END INTERFACE
```

#9) Just what is COM?

COM is an acronym. It represents the words "Component Object Model".

The short answer is that COM defines a way to communicate between modules of code. The slightly longer answer follows.

You should know that every object created or defined by this proposal is fully compatible with the COM specification. Many popular compilers are not able to make that claim accurately. The COM specification defines a standardized method of communication between modules of code (frequently called components), regardless of the platform or the tool used to create them. COM components are reusable chunks of code and associated data, which may be accessed by other "COM-Aware" components and applications.

One of the most frustrating things about this technology has been the ever-changing list of buzz-words used to describe it. We've evolved through OLE, VBX, and OCX, to COM, ActiveX, and more. Though nuances of difference abound, the important thing to remember is that COM and ActiveX describe a means of accessing code and data located outside of the current module. COM+ refers to some extensions which are specific to Win2000, WinXP, and WinVista. Throughout this discussion, we'll use the terms COM Object and ActiveX Object to describe components: reusable chunks of code and associated data.

The current versions of PowerBASIC introduced client COM services, which were accessible through the COM DISPATCH interface. While the DISPATCH interface is very flexible and easy-to-use, that very flexibility adds a level of overhead which is

unacceptable for many applications. The future could bring the capability to create and access COM objects through a DIRECT INTERFACE or a DISPATCH INTERFACE.

All objects in PowerBASIC, COM or not, present or future, will follow all the guidelines and implementation rules established for COM Objects. This simplifies usage by the programmer, yet adds no measurable overhead at run-time. PowerBASIC would encapsulate all the low-level details of the actual COM communication process. This provides a straightforward way to load and communicate with a COM component using straightforward BASIC syntax. As usual, you'll find that any PowerBASIC object implementation will be very efficient, with virtually no degradation of execution speed as compared to standard Subs and Functions. "Smaller, Faster" will always be the primary guideline of PowerBASIC.

#10) What is a COM component?

A COM component is commonly referred to as a COM Object. We can visualize a COM component or Object as simply a "black box" that comprises a set of methods and associated data. Internally, these Objects contain reusable code (Methods), and provide ways for an application to call the object's Methods and read/write its associated data through its Interfaces. Notice that this is the same definition as an object internal to your program. The difference is that COM offers a way to perform this functionality on an object external to your program.

A COM Component is generally known as a COM SERVER, because it serves up information or actions requested by a COM CLIENT. A COM SERVER makes its Methods and Properties public, so that a COM CLIENT can call them as needed.

COM Components usually take the form of an EXE, or DLL/OCX file, but the actual file extension is largely irrelevant. However, DLL/OCX versions of a component are generally referred to as "in-process", since they are loaded into the address space of the calling application. EXE-versions are typically "out-of-process" because they will not share the address space of the calling application.

To summarize, a COM Object (COM Server) is a special form of code library (similar to a standard DLL) that conforms to the COM specification. It provides at least one public interface, and is identified by a globally unique PROGID and CLSID.

Every class is associated with a GUID (a 128-bit number or string) which uniquely identifies this particular class from all others, anywhere in the world. This identifier is called the Class ID, or CLSID. A friendlier version of the CLSID is a shorter text name, which also identifies the Class. This text name is known as the PROGID, though it's possible this PROGID may not be totally unique. As it's a simpler construct, it might be duplicated in another program. These identifiers are stored in the Windows Registry when the COM component is installed and registered. PowerBASIC programmers would reference COM components by their PROGID string, and rarely by their CLSID. However, since these two items are stored in pairs, it is straightforward to retrieve the matching PROGID for a known CLSID, and vice versa.

As mentioned earlier, you don't need to know how a television works internally to use it and benefit from it. Likewise, you don't need to know how a COM Object works internally to use it and benefit from it. You only need to know the intended job of the object, and how to communicate with it from the outside. The internal workings are well "hidden", which is called encapsulation. Since we aren't able to "look inside" a

COM Object, it's not possible to directly manipulate internal variables or memory. This provides a increased level of security for the internal code and data.

So, how do you publish a COM component?

Publishing an object means making it available for access and use by other applications through the facilities of the COM Services of Windows. With some compilers, this requires pages upon pages of code. With PowerBASIC, you'll likely find it's fairly straightforward. Just add a CLSID GUID and the words "AS COM" to the end of the CLASS statement. Then, add an Interface ID (IID) to the end of the INTERFACE statement. Believe it or not, that's just about it!

Code:

```
$MyClassGuid = guid$("{00000099-0000-0000-0000-000000000008}")
$MyIfaceGuid = guid$("{00000099-0000-0000-0000-000000000009}")

CLASS MyClass $MyClassGuid AS COM
  INTERFACE MyInterface $MyIfaceGuid
    INHERIT IAutomation
    METHOD Method1(parm AS LONG)
      CALL abc(parm)
    END METHOD
  END INTERFACE
END CLASS
```

Wouldn't you rather have the compiler handle all the messy details of COM for you? The name of the CLASS (in this case MyClass) will be used as the ProgID for COM registration of the DLL. The GUID's you selected will be used for the CLSID and IID, so you're ready to go...

#11) How are GUID's used with objects?

A GUID is a "Globally Unique Identifier", a very large number which is used to uniquely identify every interface, every class, and every COM application or library which exists anywhere in the world. GUID's identify the specific components, wherever and whenever they may be used. A GUID is a 16-byte (128-bit) value, which could be represented as an integer or a string. This item is large enough to represent all the possible values needed.

The PowerBASIC GUID\$() function (or a hot-key in the PowerBASIC IDE) can generate a random GUID which is statistically guaranteed to be unique from any other generated GUID.

When a GUID is written in text, it takes the form:

Code:

```
{00CC0098-0000-0000-0000-0000000000FF}
```

When a GUID is used in a PowerBASIC program, it is typically assigned to a string equate, as that makes it easier to reference.

Code:

```
$MyLibGuid = guid$("{00000099-0000-0000-0000-000000000007}")
$MyClassGuid = guid$("{00000099-0000-0000-0000-000000000008}")
$MyIfaceGuid = guid$("{00000099-0000-0000-0000-000000000009}")
```

Every COM COMPONENT, every CLASS, and every INTERFACE is assigned a GUID to uniquely identify it, and set it apart from another similar item. As the programmer,

you can assign each of these identifiers, or they will be randomly assigned by the PowerBASIC compiler.

When you create objects just for internal use within your programs, it's common to ignore the GUID's completely. PowerBASIC will assign them for you automatically, so you don't need to give it a thought. However, if you plan to publish an object for any external use through COM services, it's very important that you assign an explicit identifier to each item in your code. Otherwise, the compiler will assign new identifiers randomly, every time you compile the source. No other application could possibly keep track of the changes.

The APPID or LIBID identifies the entire application or library. You specify this item with the #COM GUID metastatement:

Code:

```
#COM GUID $MyLibGuid
```

The CLSID identifies each CLASS. You specify this item in the CLASS statement:

Code:

```
CLASS MyClass $MyClassGuid AS COM
    ....
END CLASS
```

The IID identifies each INTERFACE. You specify this item in the INTERFACE statement:

Code:

```
INTERFACE MyInterface $MyIfaceGuid
    ....
END INTERFACE
```

Much more to follow... Stay tuned.

#12) What is inheritance?

Inheritance is all about code reuse. You can reuse the definitions of an interface, or you can reuse complete sections of code.

INTERFACE INHERITANCE is defined by COM standards, and available for use by any COM object. This form of inheritance applies only to the definition of each item in an interface, rather than the underlying code. Interface inheritance gives you the option to use one interface in multiple classes (objects). Because the interface definition remains identical in each instance, you can often use the identical (or similar) code to manipulate different objects. With this form of inheritance, the programmer must provide appropriate code for each of the Methods and Properties in every implementation of the interface.

IMPLEMENTATION INHERITANCE is the process whereby a CLASS derives all of the functionality of an interface implemented elsewhere. That is, the derived class now has all the methods and properties of this new, extended version of a Base Class! This form of inheritance is offered by PowerBASIC, even though it is not required by the COM Specification.

You can extend the functionality of an interface you created earlier by adding new methods and properties to the derived interface/class. The syntax for adding extra methods (not in the Base Class) is the same as adding methods to a standard class -

- just add methods and properties, as always.

You can add to, or replace, the functionality of a particular method or property by coding a replacement which is preceded by the word **OVERRIDE**. The overriding method must have the same name and signature (parameters, return value, etc.) as the one it replaces. When you implement a new method in a derived class, you may call a method in the Base Class by using the pseudo-object **MYBASE**. This allows you to extend the original functionality, or replace it entirely.

Inheritance is implemented by use of the **INHERIT** statement within an **INTERFACE / END INTERFACE** block. The word **INHERIT** is followed by the class name and interface name of the code to be inherited. Both are necessary, because COM allows you to have multiple implementations of any particular interface.

Code:

```
CLASS MyClass
  INTERFACE MyFace
    INHERIT IDispatch
    METHOD aaa()
      ' code...
    END METHOD
    METHOD bbb()
      ' code...
    END METHOD
    METHOD ccc()
      ' code...
    END METHOD
    METHOD ddd()
      ' code...
    END METHOD
  END INTERFACE
END CLASS

CLASS TheClass
  INTERFACE TheFace
    INHERIT MyClass, MyFace
    OVERRIDE METHOD bbb()
      ' new code
    END METHOD
    OVERRIDE METHOD ddd()
      ' new code
    END METHOD
    METHOD xxx()
    END METHOD
  END INTERFACE
END CLASS
```

Note that the derived interface "TheFace" first inherits **IDISPATCH**, and then, all four methods from "MyFace" (aaa,bbb,ccc,ddd). However, because of the **OVERRIDE** statements, both **bbb()** and **ddd()** are replaced by newer versions of these methods. Note that a derived class may be inherited by yet another class, repetitively. The depth of this inheritance is limited only by available memory.

The pseudo-object **MYBASE** may be used within a derived class to access a method in the original base class. For example, if you placed:

Code:

```
MyBase.bbb()
```

in the above derived code, it would execute the method bbb() in the parent interface/class. You could then use the results to extend or modify actions in your newer code.

#13) So how do you create an object?

This operation is frequently known as "Creating an INSTANCE of an OBJECT." Yes, this is just one more buzz-word -- but you'll hear it frequently.

In order to create an object, you first need an OBJECT VARIABLE. This object variable can be located most anywhere in your program, and have any scope: LOCAL, GLOBAL, THREADED, etc. This object variable is declared by using the name of the interface you wish to access on the object. This is done so that PowerBASIC knows which Methods can be called via this variable. This variable is expected to be a "container" for an OBJECT REFERENCE (that is, a pointer to the actual object). Initially, this variable is automatically set to "NOTHING". If you wish to use the generic DISPATCH interface to access the object, you would use the name DISPATCH instead.

```
LOCAL object1 AS MyInterface  
LOCAL object2 AS DISPATCH
```

There is actually one more special case, that of an IDBIND DISPATCH interface. Since object creation works the same on those interfaces, as well, we'll have more on that special topic in a later section. So, now that you have two empty object variables, what do you do with them? Use the assignment statement (LET) to create an object!

To create an object, you need to specify a CLASS and an INTERFACE. The interface is implied by the object variable you use, so it only remains that you specify the CLASS name. If the requested CLASS is internal to your program, use the word CLASS:

```
LET object1 = CLASS "MyClass"
```

The class name ("MyClass") must be specified as a quoted string literal, which is the name of a class implemented within the program. Since the class is internal (the name is known at compile-time), you may not use a string variable or expression. Upon execution, a new object is created, and a reference to that object is assigned to the object variable object1. The interface requested is determined by the original declaration of object1. If the interface name is DISPATCH, you can call the methods with the OBJECT statement -- otherwise, regular Method and Property references are used for direct interfaces.

```
LET objvar = NEWCOM ProgID$  
LET objvar = GETCOM ProgID$  
LET objvar = ANYCOM ProgID$
```

This form of the LET statement is used to obtain an object reference external to the program using the COM facilities of Windows. If the requested object is in a DLL (an in-process server), you will always use the NEWCOM option, as you're asking Windows to supply a new object. If the request is successful, an OBJECT REFERENCE

(a pointer to the object) is assigned to the objvar.

If the requested object is in an EXE (out-of-process server), you may use any of the three options. If the director word NEWCOM is specified, a new instance of a COM application is created. With GETCOM, an interface will be opened on an existing, running application, which has been registered as the active automation object for its class. With ANYCOM, the compiler will first try to use an existing, running application if available, or a new instance if not.

Of course, as with any other LET (assignment) statement, you are free to simply omit the word LET entirely.

If an object creation or assignment fails for any reason, the object variable is set to NOTHING. If this statement fails, no errors are generated, nor is an OBJRESULT set. You should test for success of the operation with ISOBJECT(objvar) before trying to use the object or execute its methods.

But what about the rare case when there's no ProgID\$ available? There's an answer for that, too.

```
LET objvar = NEWCOM CLSID ClassID$  
LET objvar = GETCOM CLSID ClassID$  
LET objvar = ANYCOM CLSID ClassID$
```

This new form also obtains a COM object reference, just as in the previous example. However, it is only used in the unusual case of a COM Object which has no ProgID. It works exactly as the original form above, except that it describes the requested object by its 16-byte GUID which is the ClassID of the object.

```
LET objvar = NEWCOM CLSID ClassID$ LIB DLLPath$
```

PowerBASIC offers the unique ability to create and reference COM objects without any reference to the registry at all. As long as you know the CLSID (Class ID) and the file path/name of the DLL to be accessed, you can do so with no registry access at all. You don't need a special type of COM server. This technique can be used with any server, whether created by PowerBASIC or another compiler. By using this method of object creation, there is simply no need for the server to be registered at all. That allows you to keep local copies of the COM servers you use, with no chance they will be altered or replaced by another application. You use the above form, where the clause "CLSID ClassID\$" identifies the 16-byte Class ID, and the clause "LIB DLLPath\$" identifies the file path and file name of the COM Server. Once you've obtained the COM object reference in objvar, it is used exactly as you would with a traditional object.

#14) How do you duplicate an object variable?

In the previous section, you learned to create an object, which assigns an OBJECT REFERENCE to the object variable:

Code:

```
LOCAL object1 AS MyInterface  
LET object1 = CLASS "MyClass"
```

What if you need to duplicate it? Well, you first must decide whether you want to create a completely new object, or if you just want a second object variable which points to the the same object. This is a very important distinction. With two objects, they each have their own set of INSTANCE variables. The variables in each set remain independent of the other set until they are destroyed. You would create two objects by writing:

Code:

```
LOCAL object1, object2 AS MyInterface
LET object1 = CLASS "MyClass"
LET object2 = CLASS "MyClass"
```

If you have two object variables pointing to the same object, they would share the same set of INSTANCE variables. You would create two OBJECT REFERENCES TO ONE OBJECT by writing:

Code:

```
LOCAL object1, object2 AS MyInterface
LET object1 = CLASS "MyClass"
LET object2 = object1
```

Of course, now we can take this one step further. You already know that an OBJECT may have two (or even more) interfaces defined in a CLASS. How would you actually use two interfaces on the same object? Just declare an object variable for each interface, much like:

Code:

```
LOCAL object1 AS MyInterface
LOCAL object2 AS HisInterface
LET object1 = CLASS "MYClass"
LET object2 = object1
```

The code is very much like the preceding example, except that the two object variables are declared as two different interfaces. When the last line is executed, PowerBASIC looks at the object variables to determine if they represent the same interface or not. If they do, it simply creates an extra variable, pointing to the same object. If they differ, PowerBASIC checks object to ensure the new interface is supported. If so, it creates a new OBJECT REFERENCE via the new interface, and assigns it to object2. It's just that simple!

The final issue in this topic is how to destroy an object variable. Generally speaking, you do nothing at all. When an object variable goes out of scope, PowerBASIC will handle all the messy details for you. For the most part, just forget about it. However, in the rare case that you need to destroy an object variable at a specific time and place, you can do so with the following statement:

Code:

```
object1 = NOTHING
```

Setting an object variable to NOTHING handles it all for you.

#15) How do you call a direct method?

First, you should remember that INSTANCE variables may only be accessed from within the object. The only way to access them from the "outside", is by a parameter or return value of a METHOD or PROPERTY function. Of course, Methods and Properties may also utilize the other data scopes: Global, Local, Static, and Threaded.

In PowerBASIC, the basic unit of code in an object is the METHOD. A METHOD is a block of code, very similar to a user-defined function. Optionally, it can return a value, like a FUNCTION, or merely act as a subroutine, like a SUB. Methods are implemented when you write:

Code:

```
METHOD name [alias "altname"] (var as type...) [AS type]
    statements...
    METHOD = expression
END METHOD
```

Methods can only be called through an object variable, which is an integral part of the calling syntax. The object variable must be valid, that is, it must contain a valid object reference which was assigned to it with the LET statement. If you attempt to call a method on a null object, you'll likely experience a GPF and a total failure of your program. Methods may be called by writing:

Code:

```
Dim ObjVar as MyInterface
Let ObjVar = Class "MyClass"

[CALL] objvar.Method1(param)
```

Note the word CALL is optional. This example shows how to call "Method1" when "Method1" does not return a value. If it did have a return value, use this form instead:

Code:

```
var = ObjVar.Method1(param)
```

A PROPERTY is a special type of METHOD, which is only designed to GET or SET INSTANCE data in an object. While the work of a PROPERTY could readily be accomplished with a standard METHOD, this distinction is convenient to emphasize the concept of encapsulation of INSTANCE data within an object. There are two forms of PROPERTY procedures, PROPERTY GET and PROPERTY SET. As implied by the names, the first form is used to retrieve a data value from the object, while the second form is used to assign a value. Properties are implemented:

Code:

```
PROPERTY GET name [alias "altname"] (ByVal var as type...) [AS
type]
    statements...
    PROPERTY = expression
END PROPERTY

PROPERTY SET name [alias "altname"] (ByVal var as type...)
    statements...
    variable = value
END PROPERTY
```

When you use PROPERTY SET, the last (or only) parameter is used to pass the value to be assigned. A PROPERTY may be considered "Read-Only" or "Write-Only" by simply omitting one of the definitions. However, if both GET and SET forms are defined for a particular Property, parameters and the property must be identical in both forms, and they must be paired. That is, the PROPERTY SET must immediately follow the PROPERTY GET. It's important to note that all PROPERTY parameters must be declared as BYVAL.

Properties can only be called through an object variable, which is an integral part of the calling syntax. The object variable must be valid, that is, it must contain a valid object reference which was assigned to it with the LET statement.

You can access a PROPERTY GET with:

Code:

```
Dim ObjVar as MyInterface
Let ObjVar = Class "MyClass"

var = ObjVar.Prop1(param)
```

You can access a PROPERTY SET with:

Code:

```
Dim ObjVar as MyInterface
Let ObjVar = Class "MyClass"

[CALL] ObjVar.Prop1(param) = expr
```

Note that the choice of Property procedure is syntax directed. In other words, depending upon the way you use the name, PowerBASIC will automatically decide whether the GET or SET PROPERTY should be called. In every Method and Property, PowerBASIC automatically defines a pseudo-variable named ME, which is treated as a reference to the current object. Using ME, it's possible to call any other Method or Property which is a member of the class: var = ME.Method1(param)

Methods and Properties may be declared (using AS type...) to return a string, any of the numeric types, a specific class of object variable (AS MyInterface), a Variant, or a user defined Type.