

Object-Oriented PowerScript Programming

A white paper by D. William Reynolds, Jr.

^a 1996, D. William Reynolds, Jr. and Austin Software Foundry

Basic Elements of the PowerScript Programming Language	1
Level of Abstraction	2
Data	4
Language Syntax, Semantics and Control Statements.....	6
Dot Notation	7
Pronouns	7
Variables	8
Operators	9
Expressions.....	10
Command Statements.....	10
Functions and Events	12
Subprograms	13
Modules.....	13
Assignment.....	13
Type Checking	14
Fundamental PowerScript Concepts	15
Data Types	15
Any Type	16
Enumerated Types.....	16
Standard Types.....	17
Class Types	17
Composite Types	18
Structures	18
Arrays and Strings	21
Attributes (or Properties)	22
Declarations.....	22
Scope.....	23
Access Rights	25
Messages and Methods	26
Message Direction	26
Message Access Rights.....	26
Message Name Resolution.....	27
Message Timing.....	28
Message Parameters	28
Method Computational Intensity	28
Method Location	29
Encapsulation	29
Inheritance	30
Polymorphism	31
Static and Dynamic Typing.....	32
Inclusion Polymorphism.....	33
Operational Polymorphism.....	34
System Class Library.....	34
Programming Style and Design Considerations	39
Programming Styles Supported by PowerBuilder	40
Procedural Style.....	40
Object Based Style	41

Object-Oriented Style 41
Strongly Typed Object-Oriented Style 41
Weakly Typed Object-Oriented Style 41
Difficulties of Mixed Styles 42
Guidelines for an Object-Oriented Programming Style 42
 Class Design..... 42
 Hierarchy Design 43
 Design vs. Implementation..... 43
Program Code Conventions and Standards..... 43
 Case 44
 Considerations when Naming Variables 44
Summary44

The PowerScript Language

This paper covers the basic concepts and structure of the PowerScript language, focused on using it for object-oriented programming. Object-oriented programming languages actually have a great deal in common with procedural languages, so this paper also approaches PowerScript within a framework that can be applied to any programming language. This is valuable for a couple of reasons. First, understanding a programming language's underlying principles and structure can save you considerable time and effort by providing clear guidelines for using it correctly. Second, you should be able to use this framework to compare whatever other programming language you're most familiar with to PowerScript, and ease your transition to this very powerful new language.

The paper assumes you are already familiar with the main features and syntax of PowerScript from your previous PowerBuilder programming, so it is not an exhaustive, detailed presentation of the syntax. Instead, it is designed to give you a solid understanding of PowerScript as a programming language, not just "scripts" used inside of a "GUI builder", as is often the mistaken impression of PowerBuilder and PowerScript. If you're not familiar with the language syntax, your PowerBuilder manual and online help system are the best place to turn for these specifics. Throughout the rest of the book I'll assume if you are uncertain of the syntax in an example that you'll turn to the documentation for clarification. Instead of syntax, this paper presents the broader concepts you need to work with the newest release of PowerBuilder, and lays the foundation for future discussion of the most important and often misunderstood feature of the language, object references. It also presents the basic programming styles provided in the language and helps you decide which is best suited to your task and experience level.

With the release of PowerBuilder 5.0, the PowerScript language has several new and long-awaited features. One of the most obvious is the new editor, with color coding, which can be customized in the PowerScript painter property sheets, to identify datatypes, system level functions, flow-of-control statements, comments, and literals. It will automatically indent scripts based on flow-of-control statements. These features make the language in general much easier to use. The key language enhancements are the addition of constants, changes in how you access global variables (and better detection when a global overrides a local variable), new ways to control the access to a particular variable, function name overloading, and a series of enhancements to functions and events. Most of these enhancements have been added in response to the needs of a maturing and more sophisticated programming community, and are aimed at experienced PowerBuilder and object-oriented developers. This paper will give you a place to begin as you start to assimilate these features into your everyday PowerBuilder programming repertoire.

Basic Elements of the PowerScript Programming Language

The introduction of the PowerBuilder visual development environment five years ago was a breakthrough for everyone involved in writing high level software applications (applications above the level of things like device drivers, memory management utilities, and commercial applications like word processors and spreadsheets) for the Windows environment. It shielded us from the complexity of the Windows Software Developer's Kit API and C or C++ memory management. It transformed a large portion of our programming tasks (describing our application to the computer in terms of program code - text) into "painting" tasks where we

describe our application by drawing it and letting PowerBuilder generate the code for the computer to execute. But even with the visual programming tools in PowerBuilder, we still have to write event scripts and functions in program text in order to help PowerBuilder generate all of the code necessary for a complete application. In fact, as the complexity of your PowerBuilder applications grow and you begin to use more sophisticated techniques like application partitioning and nonvisual objects, it's necessary to write more and more actual code in PowerScript. PowerScript is a relatively simple language to learn since its syntax is similar to languages most of us have learned - Basic, Pascal, C, etc. In fact, the syntax was largely modeled on Microsoft QBASIC. The simplicity of the syntax, and the visual tools provided in the PowerBuilder environment, tend to hide some of the very powerful elements of PowerScript, mainly the object-oriented features drawn from C++, Eiffel and Smalltalk. In order to uncover this power and learn to harness it, it helps to understand some basic principles and concepts underlying the PowerScript programming language. It is also very helpful to study QBASIC, C++, Eiffel and Smalltalk to develop a broader understanding of the "whys" of the PowerScript language. We'll start by examining the seven basic elements of a programming language, and how they are implemented in PowerScript.

Level of Abstraction

All programming languages provide some level of abstraction above the level at which the computer actually executes instructions. I'll never forget one of my first programming classes at The University of California at Berkeley. My professor introduced us to programming by having us write a simple program IN BINARY on a DEC PDP-11. We actually had to key the program in with toggle switches where up meant 1 and down meant 0. That was tedious! Next, he taught us assembly language and then C. It was one of the hardest courses I took but probably the most valuable. The binary routines we wrote were directly implemented by the computer and gave us complete control over things as basic as how the computer booted itself, but would have been impossible to use to write a business application. Binary programs like this could be considered first generation programming languages. Assembler, a second generation language, was an improvement in programming efficiency and still gave us very precise control over the use of the computer's internal registers and memory structure. The problem was, except for tasks that required this level of control, it still wasn't terribly efficient to program in assembler. Our next language, C, still provided access to many of these low level aspects of the computer, but gave us higher level abstractions like data types (for example float, char and int), structures, arrays, flow-of-control statements and functions. These made it much more efficient to structure and write applications for more general problems like business systems, word processors and even compilers. C and other languages like Fortran, COBOL, and Pascal are considered to be third generation languages - three steps removed from the hardware.

Each level of abstraction away from the computer has two consequences. First, it improves the efficiency of the programmer using it by providing language elements that automate lower level functionality. Second, details are lost as higher level abstractions are used. This is an obvious tradeoff, the efficiency of the programmer versus access to low level details of the computing environment. The trick is to optimize this tradeoff. Those of us building business applications with PowerBuilder certainly need to optimize our development time and likely don't need access to bit level memory manipulation. Where we do, PowerBuilder provides access to lower level languages like C and C++ through external function calls.

Like C++ and other object-oriented languages, PowerScript provides one very important abstraction beyond C, Fortran, COBOL and Pascal. It provides the ability for programmers to define their own new data types. Integers, doubles and floats are useful data types (I'll refer to these as standard data types), or abstractions. A data type is simply a set of values and a set of operations on those values. An integer is a number and you can add, subtract, multiply and

divide them. But these standard data types alone are not sufficient abstractions for all problems encountered by programmers. In fact, it would be unrealistic to expect language designers to be able to come up with all of the abstractions needed by programmers. Instead, object-oriented languages allow programmers to define their own data types, or abstractions, using yet a higher level abstraction from procedural languages - a class. A class is created by combining lower level abstractions like floats, integers and functions. A class used at runtime is called an object and is similar to a variable, it's just more complex. Like a variable that is created from the standard data types integer or string and used at runtime to hold actual data, a class is used to define objects for use at runtime that hold actual data. We'll spend lots of time discussing how and when to use classes and objects through the rest of the book. An example of a simple class definition (the definition of a new data type) is shown in the following listing of exported PowerScript syntax. This definition is very similar in content, if not exact syntax, to the definition of a class in C++, Smalltalk or Eiffel. The example is the definition of a "book" data type for an on-line library catalog application.

```
001 forward
002 global type n_book from nonvisualobject
003 end type

004 end forward
005 global type n_book from nonvisualobject
006 end type
007 global n_book n_book

008 type variables
009 Private:

010 Protected:
011 stringis_title
012 stringis_author

013 Public:
014 end variables

015 forward prototypes
016 public function boolean nf_set_title (string as_new_title)
017 public function boolean nf_set_author (string as_new_author)
018 end prototypes

019 public function boolean nf_set_title (string as_new_title);IF NOT
IsNull(as_new_title) THEN
020 is_title = as_new_title
021 RETURN TRUE
022 ELSE
023 RETURN FALSE
024 END IF
025 end function
```

```
026 public function boolean nf_set_author (string as_new_author);IF NOT
    IsNull(as_new_author) THEN
027 is_author = as_new_author
028 RETURN TRUE
029 ELSE
030 RETURN FALSE
031 END IF
032 end function

033 on n_book.create
034 TriggerEvent( this, "constructor" )
035 end on

036 on n_book.destroy
037 TriggerEvent( this, "destructor" )
038 end on
```

• Listing 1. PowerBuilder class definition of n_book.

There is a tendency when first learning a programming language to focus on how to make the language perform actions using its statements and commands, especially in procedural languages. These are certainly important abstractions that a higher level language provides for us. After all, the goal of a program is to do something, right? But in object oriented programming languages, statements and commands are used to manipulate the data used to represent another abstraction, some new object. And that's the critical leap to make, using objects as the primary abstraction for designing and structuring your programs. It's pretty hard to design a program using an abstraction you're unfamiliar with, so studying the lower level aspects of the language that will allow you to structure how these objects are represented is where we need to focus next.

Data

What do I mean by "data" in the context of a programming language? Webster's Dictionary defines datum (the singular form of data) as "a fact on which reasoning is based". In the context of a computer or a programming language we often use the term data to mean the electronic representations of facts, that we can then apply various operations to. In order to store facts, or data, in the computer we have to identify what type of data must be stored so the computer knows how to store it, and what operations can be applied to it so the computer knows what it can do with that data. In PowerScript an integer is a type of data that can take on a finite set of values from -32,768 to +32,767, uses 16-bits of memory, and can have a set of operations applied to it including addition, subtraction, multiplication, and division. The book class (n_book) in the previous listing is a type of data that can take on an infinite combination of values for title and author, takes 1,400-bytes of memory, and can have operations applied to it to set new author and title values. So, two of the most important concepts related to data in a programming language are the type of data to be stored, and the operations that can be applied to it. I'll discuss the types of data in more detail later in the paper.

There are essentially five categories of data used in PowerScript; literals, variables, constants, enumerations, and composites. **Literals** are specific values used directly in the body of your

programs such as 365, 3.14159, 'x', FALSE, or "Joe". The value of a literal is fixed. A **variable** is a name given to the spot allocated in memory to hold a value of a specific type of data like an integer or a string. The value in that spot can change during the course of a program. A **constant** is the name given to a variable that cannot change value during the course of the program. The release of PowerBuilder 5.0 brings a new keyword, CONSTANT, to your PowerBuilder programming repertoire. You can use this keyword to modify any declared variable of a standard or enumerated type to be a constant. Doing so changes it from a variable that is evaluated at runtime, as all were formerly, to one that is evaluated at compile time. After you declare a constant, you can use it anywhere you would use a value; but if you try to assign to it, the compiler will flag it as an error. An **enumeration** is a variable that can be assigned a fixed set of values. Values of enumerated data types always end with an exclamation point (!). An example is the WindowType attribute of a window in PowerBuilder. It can only take on the finite set of values [Main!, Child!, Popup!, Response!, MDI! and MDIHelp!]. Finally, a **composite** is a combination of one or more variables, constants or enumerations. They can hold any amount and type of data defined by the programmer. Arrays, structures and objects are examples of composite data. They typically have certain unique operations that can be applied to them to make them more convenient to use. The next listing shows examples of these five categories.

```
001 //"Ernest" is a Literal
002 IF is_author <> "Ernest" THEN RETURN

003 //is_author and ii_num_pages are Variables
004 stringis_author
005 integer      ii_num_pages

006 //IS_HOMECITY and IR_PI are Constants
007 constant string IS_HOMECITY = "Austin"
008 constant real IR_PI = 3.14159265

009 //ie_toolbaralignment and ie_windowstate are Enumerations
010 ToolBarAlignment ie_toolbaralignment
011 ie_toolbaralignment = AlignAtTop!
012 WindowState = ie_windowstate
013 ie_windowstate = Normal!

014 //iwi_main and inv_book are Objects
015 w_main iwi_main
016 iwi_main = CREATE w_main

017 n_book inv_book
018 inv_book = CREATE n_book
```

• Listing 2. Five categories of data used in PowerScript.

The last example in the listing is probably the newest to you and is what distinguishes object-oriented languages from procedural languages. We'll spend much more time on it throughout the rest of the book because it's not quite this simple. All of the categories discussed above can be used to create a composite data type. A class definition is the most complex form of composite data because you can define new operations on it (functions and events), which you can't do for arrays and structures. The class definition is then used to declare and create an

object, as in the last example in the previous listing. Once you have created objects in your program you can then manipulate them using their operations. In order to do this you'll need some commands and statements to control the flow of your program.

Language Syntax, Semantics and Control Statements

The syntax of a programming language is a set of rules that define what sequences of symbols are considered to be valid expressions, or programs, in the language. This set of rules is expressed using a formal notation. I have often remarked to students that PowerScript is almost "syntax-less". What I mean is that the set of rules for expressions is very small. The release of PowerBuilder 5.0 may cause me to think twice before I make that remark again, because it does introduce a significant number of new rules to the language. Even so, the syntax of almost any programming language is the easiest part to learn and I still believe that the PowerScript syntax is simple and straightforward. It's the semantics of the language that can be difficult to learn.

By semantics, I am referring to the meaning of an expression in the specific context of a program. "The bull is in the pen." and "The ink is in the pen." require a deeper understand of English than just sentence structure (syntax). The context and underlying meaning are essential to understanding which type of "pen" is being referred to. At any point in time during its execution the "context" of a program can be described by its state (the current contents of memory and the next instruction that's about to be executed). That next instruction will change the state of the program, but will it result in a correct state and exactly what are all of the implications of this change to the program's state? It's necessary to understand what's in memory and how the next instruction will affect memory to be sure that the state change will result in the desired new state. In the following example listing, what do you need to know to figure out what will happen at runtime when lines 002 and 003 are executed? This question has to do with object-references, the subject of another paper.

```
001 //Clicked event for m_checkoutbook
002 inv_book = inv_library_item
003 inv_book.TRIGGER DYNAMIC FUNCTION CheckOut( )
```

• Listing 3. PowerScript semantics .

The Eiffel object-oriented programming language actually has a syntax for testing the state of a program and ensuring that an instruction started and ended with a valid state. These are called "assertions" in Eiffel and result in extremely robust programs. They can help insure that in situations like the one in the example above the program is in a valid state to execute those lines of code. We don't have assertions in PowerBuilder, but with a solid understanding of what's going on behind the scenes and some programming discipline you can program this kind of reliability into your PowerBuilder applications!

PowerScript language syntax consists of 14 basic elements; comments, identifiers, dot notation, labels, reserved words, pronouns, statement continuation and separation, white space, variables, operators, expressions, command statements, functions and events, and SQL statements. I don't plan to cover all of these elements. Many of them are straightforward and presented well in the product documentation on the PowerScript language. I will cover only those that are either key to using the object-oriented features of PowerBuilder or are not presented in the documentation.

Dot Notation

Dot notation is a technique used in object-oriented languages in order to extend procedural style variable and function names so they can be referenced from outside of an object they are declared within. A variable or function may be preceded by one or more object names, each separated by a "dot". An example is a control, `cb_OK`, inside of a main window, `w_main`. This control contains a variable called "Text" and a function called "Resize(width, height)". A program or object outside of the window this control is displayed on can access these members of `cb_OK` with the following dot notation:

```
001 w_main.cb_OK.text = "Save"
002 w_main.cb_OK.resize(400, 100)
```

- Listing 4. PowerScript dot notation .

The attribute (or property) and function "Text" and "Resize(width, height)", respectively, are said to be *members of* `cb_OK`, and `cb_OK` is said to be a *part of* `w_main`.

In PowerBuilder 5.0 dot notation has been extended to DataWindow objects, both within DataWindow controls and within the new DataStore nonvisualobjects. A DataStore is nothing more than a DataWindow object embedded within a nonvisualobject. The nonvisualobject acts like a wrapper around the DataWindow object, just like a DataWindow control does. In order to reference a specific value in a DataWindow object inside either a control or nonvisualobject you can now use the following dot notation syntax:

```
001 //Access a data value within a DataWindow object
002 dwcontrol.Object.columnname.buffer.datasources
003 datastore.Object.columnname.buffer.datasources

004 //Access a nested object within a DataWindow object
005 dwcontrol.Object.objectname.attribute
006 datastore.Object.objectname.attribute
```

- Listing 5. PowerScript dot notation for DataWindow and DataStore object .

Pronouns

There are four pronouns in PowerScript - THIS, PARENT, PARENTWINDOW, SUPER. They are described in the PowerScript documentation as generic references to objects. Additionally, it is important to recognize that they are actually expressions that are evaluated (computed) each time they are encountered in a program. The resulting value they return is a reference to an object. THIS computes a reference to the object itself. PARENT computes a reference to the object that contains the object itself. PARENTWINDOW computes a reference to the window that contains a Menu object. SUPER computes a reference to an object's immediate ancestor. These references can be used directly;

```
001 Close(THIS)
```

- Listing 6. PowerScript pronouns .

or assigned to a variable and used in subsequent expressions without recomputing them.

```
001 window lwi_this
002 lwi_this = THIS
003 RETURN lwi_this
```

- Listing 7. PowerScript pronouns in expressions .

The implication of the second usage is that it will always be more efficient to use the pronouns to compute a reference once, assign it to a variable, then reuse that variable throughout a given script. PowerBuilder then doesn't have to recompute the value of the pronoun every time it encounters it separately in the program. There are also other implications to this characteristic of pronouns that we'll cover in a future publication.

Variables

Variables within PowerBuilder are one of the significant discussion in a future publication, but I'll review a few aspects of them here along with a new feature in PowerBuilder 5.0. The key to preparing yourself for the discussion of references is to remember that variables are nothing more than a name for a spot in memory. That spot may contain an integer, a string, an object, or any other legal data type in PowerBuilder. Variables in PowerBuilder can have one of four levels of scope; local, instance, shared or global. Variables with instance scope are declared within a class definition and are "owned" by each instance, or object, of that class declared at runtime in the program. These instance variables have an additional property attached to them, access rights. This property is discussed in more detail later in this paper, but briefly, an instance variable can have public, protected or private access rights. Public access rights means the variable is visible to all other objects, protected access rights means it is visible only to descendants of the object where it is declared (members of that object's "family tree"), and private access rights means that no script outside of the object where the variable was declared has access to it. PowerBuilder 5.0 has added a new twist to variable access, however.

Access rights refers to the visibility of an instance variable outside of the class it is declared in. The new property of instance variables is read/write access. In addition to controlling the visibility of an instance variable, you can now control modifications to the value(s) it contains. Public instance variables can be declared to have PROTECTEDREAD/PROTECTEDWRITE or PRIVATEREAD/PRIVATEWRITE. Protected instance variables can be declared to have PRIVATEREAD/PRIVATEWRITE. Private instance variables don't need any further protected since they aren't visible anyway. These keywords behave very similar to the PROTECTED and PRIVATE access rights keywords. For example, PROTECTEDREAD means only scripts for the object and its descendants can read the variable. PROTECTEDWRITE means only scripts for the object and its descendants can change the value of the variable. Thus the following declaration of an instance variable:

```
001 Public:
002     PROTECTEDWRITE string is_customer
```

- Listing 8. Instance variable declarations .

would result in all scripts in an application being able to access `is_customer`, but only scripts in the customer object or its descendants could modify its value. I don't recommend the use of these new keywords in a fully object-oriented application, since they encourage you to allow full or partial violations to the principle of encapsulation. They are one of those language features provided by Powersoft for programmers who are not comfortable yet with the object-oriented programming style. We'll discuss the various programming styles that PowerBuilder can support later in the paper and I'll make some recommendations of things to use and to avoid with each style.

Operators

The operators provided in PowerScript are relatively straightforward and fall into 5 categories: arithmetic (+, -, *, /, ^), logical (=, <, >, <>, <=, >=), relational (NOT, AND, OR), concatenation (+) and navigation (*object::member*, *object nestedobject*, *::globalvariable*). Except for the navigation operators, which are difficult to find any reference to in the documentation and are related exclusively to object-oriented programming, the operators provided by PowerScript are relatively standard and should not be a source of mystery or confusion. The navigation operators, on the other hand, may be somewhat mysterious.

The three navigation operators in PowerScript are referred to as membership (*object::member*), nested object (*object nestedobject*) and global scope (*::globalvariable*) operators. They all provide ways to navigate among references to objects. The first two provide mechanisms for navigating among related objects in an inheritance hierarchy or in an aggregation, respectively. The global scope operator provides a mechanism for navigating among object references in an application. The membership operator separates an object reference from a member function or event. The nested object operator separates a parent object reference from a child object reference in an aggregation. The global scope operator forces PowerBuilder to use a global variable even if there is a local or shared variable with the same name. The only acceptable use of a global variable in object-oriented programming is as a reference to an object, so this operator will allow you to force PowerBuilder to skip over local and shared object references to a global reference. The examples in the following listing should give you some ideas of the combinations that are possible:

```
001 /*****
002 Script in the constructor of the commandbutton w_dialog`cb_1
003 inherited from w_main`cb_1
004 *****/
005 //Referencing a member function in the ancestor. The following
006 //statements achieve identical results.
007 w_main`cb_1::uf_settext( )
008 w_main.cb_1.uf_settext( )
009 w_main.cb_1.FUNCTION uf_settext( )
```

```

010 SUPER::uf_settext( )
011 //SUPER.uf_settext( )           (this statement does not compile)

012 //Referencing a member event in the ancestor.  The following
013 //statements achieve identical results.
014 w_main`cb_1::TriggerEvent("clicked")
015 w_main.cb_1.TriggerEvent("clicked")
016 w_main.cb_1.EVENT clicked ( )
017 SUPER::TriggerEvent("clicked")
018 //SUPER.TriggerEvent("clicked")(this statement does not compile)
019 CALL w_main`cb_1::clicked

020 //Referencing a member attribute in the ancestor.  The following
021 //statements achieve identical results.
022 string ls_text
023 //ls_text = w_main`cb_1::Text (this statement does not compile)
024 ls_text = w_main.cb_1.Text
025 //ls_text = SUPER::Text       (this statement does not compile)
026 //ls_text = SUPER.Text        (this statement does not compile)

```

• Listing 9. PowerScript navigation operators .

It's important to notice that the only consistent way to reference members of objects is through dot notation as in lines 008, 009, 016, 017, 026.

Expressions

An expression is made up of a combination of one or more literals, variables, operators, pronouns and functions or events. An example of an expression is *counter + 1*. PowerBuilder will evaluate this expression and add 1 to the value stored in the variable location named *counter*. They can occur in PowerBuilder in many places: assignment statements (*counter = counter + 1*), boolean expressions in IF...THEN statements, limits of FOR...NEXT and DO...WHILE loops, parameters of functions or events, DataWindow column validations, etc. The important concept beyond the syntax of expressions is that wherever an expression is located, it is evaluated and produces a *value*. The value is then used based on the context of the expression. In an assignment statement, the value of the expression is assigned to a variable. In an IF...THEN statement the value is used to determine how to execute the IF...THEN statement. This characteristic that an expression produces a value will be important when I discuss assignment statements below. Beyond that, expressions in PowerScript are similar to expression in C, C++, Pascal, etc. and are relatively straightforward.

Command Statements

PowerScript provides four categories of command statements: assignment, object management, control and SQL statements. I will discuss the assignment statement below. SQL statements are largely beyond the scope of this book and I presume you are familiar with Structured Query Language. Control statements can be further broken down into three categories of their own: jump, choice, and loop statements. Jump statements include CALL

and GOTO and are discouraged in both structured (or procedural) and object-oriented programming styles. Choice statements include the forms of IF...THEN and CHOOSE...CASE statements. Loop statements include FOR...NEXT and DO...WHILE statements. These are standard implementations and should be familiar to you from other programming languages. One note about choice statements that you should observe as you are expanding your use of object-oriented programming is that there should be fewer and fewer of them in your code as you learn to take advantage of polymorphism. This is one of the primary characteristics of polymorphism. It is a mechanism built into an object-oriented language for automatically making "choices" among alternate forms of operations, so you don't have to do it within your scripts. If you find yourself programming extensive IF..THEN or CHOOSE...CASE statements to determine the type of an object you want to operate on, this is a sure sign that you should revisit your design and that it is not very object-oriented.

Finally there are the object management statements CREATE and DESTROY. The CREATE statement allocates memory for, and creates in that memory space, an instance of a class (n_book below). The CREATE statement also returns a reference to that spot in memory, which you must capture in a variable (lnv_book below) if you want to reference that instance of the class (or object) later in a script. These references are the topic a future publication. There are two forms of the CREATE statement. The first only requires that you supply the class name as a literal that you want used to define and allocate the memory space being created. An example is:

```
001 n_book lnv_book
002 lnv_book = CREATE n_book
```

• Listing 10. PowerScript object creation .

The second allows you to supply the class as a string variable at runtime. For example:

```
001 string ls_classname
002 nonvisualobject lnv_objectreference
003 ls_classname = "n_book"
004 lnv_objectreference = CREATE using ls_classname
```

• Listing 11. PowerScript object creation using a variable .

The DESTROY statement releases the memory allocated by the CREATE statement. PowerBuilder does very little automatic garbage collection for objects. If you CREATE an object, you should use the reference you obtained to destroy it when you no longer need it. In a couple of situations, even if you didn't create an object you must destroy it if you used it. The primary examples are the DataWindow DWObject and the OLEObject. Once you obtain a reference to either one of these, even though they were created by PowerBuilder, you must manually destroy them. The act of obtaining a reference causes PowerBuilder to override its automatic deletion of these objects.

```
001 DESTROY Inv_book
002 DESTROY Inv_objectreference
```

- Listing 12. PowerScript object destruction .

Once again, a thorough discussion of these issues would require a dedicated paper of their own.

Functions and Events

Even if you are new to PowerBuilder, you are very likely familiar with common syntax for functions, and their cousins, the subroutines—such constructs appear in many languages. In Pascal, for instance, a subroutine is a chunk of code that you can start from anywhere in your program (with the right scope). You can send it a set of parameters to operate with if needed. It performs the operation and passes control back to the calling code, returning no value. A function in Pascal, on the other hand, does nearly the same thing, but always returns a value. C does not have the distinction between functions and subroutines; in C, these flexible blocks of code are always called functions, whether they return values or not.

The terminology is a bit different in PowerBuilder. A function in PowerBuilder is a block of code that you call in much the same way as Pascal or C; you send it a set of one or more parameters, it performs its function, and returns control back to the code that called it. PowerBuilder functions may or may not return values. A subroutine in PowerBuilder is really just a function without a return value.

PowerBuilder also adds another way of executing code that's not in the immediate script—through an event. One key difference between events and functions prior to PowerBuilder 5.0 is that events have not taken parameters and have not returned values, but the more important difference is that they operate *asynchronously*. That is, when you call a function in a script, control returns to your script only after the function has completed its tasks. When you post an event from a script, though, control returns to your script immediately, and the code associated with the event gets executed at some unspecified time in the future. Generally, when choosing between the two, you base your decision primarily on whether the message needs to be executed immediately, whether it needs parameters to execute, and whether the caller requires a return value.

Unfortunately, one of the most controversial debates in PowerBuilder programming circles is “When should I use a function or an event?”. My observation of actual usage suggests that programmers are making this choice based on convenience rather than on the requirements of their designs. It is easier, in the PowerBuilder painters, to override an event from an ancestor class than to override a function inherited from the same ancestor. Additionally, it is easier in PowerScript to trigger an event than to call a function. Triggering an event is not evaluated by the compiler so the programmer can afford to be sloppy. What most PowerBuilder programmers don't realize is that by relying on this characteristic of events they are bypassing the typing mechanism in PowerBuilder, which we'll discuss shortly. Calling a function on the other hand is evaluated by the compiler and often requires significant design discipline to accomplish the same effect as triggering an event that may or may not exist at runtime. This is because the PowerBuilder compiler is checking the type of object that the function is being called in, and forcing the programmer to ensure that the function actually exists in the object in the correct format. These tradeoffs have huge performance implications, as well as long term object reusability implications, which we'll cover from different perspectives throughout the rest of the book. My personal guideline is still that an event is asynchronous, takes no parameters and returns no value. An event is a “stimulus” or signal sent from a user, operating system, or other

application to notify the system an “event” has occurred and it should proceed accordingly. No other information is required. A function is synchronous, accepts parameters, and returns a value. It is the primary method for communication between objects within an application and is used to implement the operations of these objects. It is the mechanism used by objects to pass information among themselves.

Subprograms

Another major element of any programming language is the subprogram. These go by many names: procedures, functions, subroutines, methods, operations, events, etc. They all have three common characteristics. They contain data declarations, executable statements, and can be invoked repeatedly from different parts of a program. When a subprogram is called it is typically passed a sequence of values called parameters. PowerScript contains two flavors of subprograms, events and functions, which we introduced in the previous section. Subprograms in PowerBuilder, or any language for that matter, should be limited to 40-50 lines of executable statements and perform a single well defined task. A subprogram to identify and parse a sentence and divide it into a number of text strings (words) is a good example.

Modules

Subprograms are a language structure used to group logically related statements together and to package them so they can be used again and again throughout a program. Modules are used in modern programming languages to group data structures and subprograms, or possibly other modules. Once again, practical guidelines for modules suggest that they shouldn't contain more than 40-50 subprograms, or other modules, and should perform a single, well defined task. A spell-checking module in a word processor is a good example. PowerBuilder provides two constructs for creating modules - classes and PowerBuilder libraries (.PBLs).

Modules provide a number of advantages over more traditional means of organizing large systems. First, data and subprograms can be encapsulated inside of a module and access can be controlled through a well defined public interface to the module. These interfaces can even be checked at compile time to prevent errors and misunderstandings between modules. Second, the process of encapsulation can hide the internal details of a module from users of the module. When internal details change, this change can be isolated so it doesn't affect the user of the module's public interface. Third, as long as the public interface to a module remains unchanged they can be modified, recompiled and/or replaced without recompiling the entire application.

One of the largest problems with modules is deciding what goes in them, and what the public interface should look like. This is where object-oriented analysis, design and programming come to the rescue. The premise of object-oriented development is that a module should be created for any real or abstract “object” that can be represented by a set of data and operations on that data. A library of these modules should represent a subsystem or category of behavior that all included modules (objects) contribute to and participate in. I have rarely seen a PowerBuilder application that takes good advantage of PowerBuilder libraries as replaceable modules. And surprisingly to me, I still see too many PowerBuilder applications that don't make good use of classes as independent, reusable modules. Read on and maybe together we can fix that!

Assignment

Assignment statements appear to be relatively simple and are of the form *variable=expression*. They are actually quite complicated and it is extremely important to understand exactly what they

are doing in order to take full advantage of the object-oriented features of PowerScript. The variable in the assignment statement can be of any valid type in PowerScript, as long as the expression evaluates to a value of that type or a compatible type. This is based on our earlier discussions of variables and expressions, and the characteristic that expressions are evaluated and result in a single value. The assignment statement (=) causes the execution of three separate tasks, in this order: compute the value of the expression on the right-hand side of the statement, compute the address of the variable on the left hand side of the statement, copy the value from the right-hand side into the memory located at the address specified by the left-hand side. This description should begin to shed some light on the following assignment:

```
001 Inv_objectreference = THIS
```

- Listing 13. PowerScript assignment statements .

The THIS pronoun is actually an expression that PowerBuilder has to evaluate and return a value for, as we discussed earlier. The value in this case is actually an address to the location in memory of the object itself. PowerBuilder then copies that address into the variable Inv_objectreference. The address held in the variable Inv_objectreference is now the same as the address of THIS.

PowerScript also includes a number of assignment shortcuts. These include ++, --, +=, -=, *=, /= and ^=. An assignment shortcut allows you to replace assignment statements like:

```
001 counter = counter * 10
002 row = row + 1
```

- Listing 14. PowerScript variable assignments .

with ones like:

```
001 counter *= 10
002 row++
```

- Listing 15. PowerScript assignment shortcuts .

Type Checking

The previous discussions of variables, expressions and assignment statements are critical to understanding type checking. In traditional programming languages type checking has been something that programmers didn't think about too much. The languages provided a relatively small number of data types, and as long as the program didn't try to assign a value to a variable of an incompatible type everything was fine. Since there were a small number of types to check there wasn't much opportunity for confusion. Object-oriented languages now allow developers to define an infinite number of user defined data types (classes). Some of those data type are subtypes of other types and some assignments may be legal in one context but not another...ouch! Type checking has now become a much more important issue.

The three step discussion of assignment statements will give us a good foundation for exploring the issue of type checking. Type checking is a check by the compiler or runtime environment that the type of an expression (the right-hand side of an assignment) is compatible with the type

of the target variable (the left-hand side of an assignment). This also includes the assignment of a variable (right-hand side) to a function argument (left-hand side) in a function call. There are three approaches to type checking that a language can take.

The first approach is for the compiler to do nothing. In this case it is the programmer's responsibility to ensure that an assignment or function call will be meaningful at runtime. This is often referred to as *weak typing* and is the approach taken by Smalltalk. This approach makes the job of writing programs easier, but makes it much more difficult to find bugs and to ensure the reliability of a program. Run time errors that are found by users can be expensive and sometimes dangerous.

The second approach is to implicitly convert the value of an expression (right-hand side) in an assignment to the type of required by the left-hand side. This approach is almost never employed universally by a language, but on a limited basis where there is extremely high reliability that the conversion is valid. Otherwise, this approach would lead to a false sense of security by programmers, and to many of the problems with weak typing.

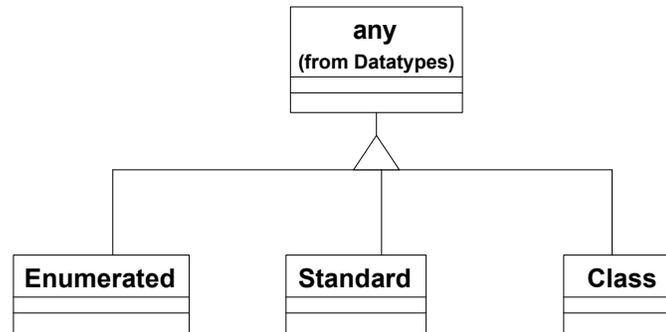
The third approach is for the compiler to refuse to execute, or even compile, an assignment if the types on both sides do not match or to allow a function call if the function doesn't exist in the target object. Remember that an object is just a user defined type. The compiler can check the "type" of an object to see if its definition (class) includes a function that matches the one being called. This is often referred to as *strong typing*. This approach results in highly reliable programs, elimination of obscure bugs, and easier integration of modules in large projects. The tradeoff is that this approach does require more programming effort to define and utilize an appropriate set of types. As a compromise for certain situation, most strongly typed languages provide mechanisms to bypass type checking when needed. PowerScript is a strongly type language, with mechanisms for bypassing type checking in certain circumstances, which we'll cover in the next section. Bypassing the type checking within PowerBuilder is not something that should be done lightly, or extensively throughout an application. The fundamental rule is that it should only be done in the context of a complete design, and not just for convenience in a particular programming situation.

Fundamental PowerScript Concepts

As I pointed out before, in object-oriented languages like PowerScript it's critical to understand how data can be structured before beginning to learn methods for manipulating that data. In the previous section I defined the basic language structures available in PowerScript and expanded on some of them that will be critical for our use of object-orientation in PowerScript. Now we'll use that introduction to dig even deeper into some key PowerScript concepts for structuring and then manipulating data.

Data Types

There are five categories of data in PowerScript as I discussed earlier: literals, variables, constants, enumerations and composites. Data in each of these data categories can take on a variety of data types. This variety of available data types can be broken down into four data type categories that do bear some relationships to each other as described in Figure 1.



• Figure 1. PowerBuilder Data Types.

Figure 1 is a conceptual representation of how the four data type categories are related to each other, not an actual physical representation of how they are implemented with PowerBuilder. In some object-oriented languages like Eiffel this is actually how the data types in the language are implemented. This conceptual representation will be extremely helpful to keep in mind as we discuss data types throughout the rest of the book.

Any Type

The Any type can only be used to for variables, enumerations, and composites. They cannot be used as literals, or constants. A variable whose type is Any takes the data type of the value assigned to it. Any variables can hold data values of enumerated data types, standard data types, class data types, and composite data types. This is why I've described the relationship between Any and the other data types like I have in Figure 1. Any variables are declared and used just like other variables. Each element of an array of Any variables can have a different data type. To determine what type of data is in an Any variable you can use the `ClassName()` and `TypeOf()` functions, and to assign the contents of an Any variable to another variable you must make sure the assignment is valid. Similarly, operations can be performed on the Any variable only if they are valid for the data type of the value contained in the Any variable. If not, you will get a runtime error which can be extremely difficult to track down. This, and the fact that Any variables are expensive to evaluate at runtime, are the two main reasons to limit your use of Any variables as much as possible.

Unfortunately, Powersoft has placed certain restrictions on the use of Any variables. The basic rule is that you can store data of any type in an Any variable, but to use it you must assign that value back to a variable of the correct data type. For example if an Any variable is a structure, you must assign the value to a structure of the appropriate type before you can access the members of the structure. We will explore these restrictions, and the use of Any variables, further in a future publication where we will have adequate space to discuss references.

Enumerated Types

Native data types are those that are built directly into a language itself, as opposed to those that are included with the language in standard or add-on libraries. PowerBuilder comes with a long list of enumerated data types defined as native types within the language. Unlike C++, where you can define your own enumerated data types, PowerBuilder does not allow you to create new enumerated types. You can however declare variables of any of the native enumerated types. A few examples of the native enumerated types are:

```

001 alignment    = {center! justify! left! right!}
002 arrangeopen = {cascaded! layered! original!}
003 borderstyle = {stylebox! stylelowered! styleraised!
                  styleshadowbox!}
004 dwbuffer     = {delete! filter! primary!}
005 exceptionaction = {exceptionfail! exceptionignore! exceptionretry!
                     exceptionsubstituereturnvalue!}
006 windowstate = {mazinged!, minimized! normal!}

```

- Listing 16. PowerScript enumerated type examples .

An example of declaring and using an enumerated variable is:

```

001 windowstate le_windowstate

002 le_windowstate = THIS.windowstate

003 CHOOSE CASE le_windowstate
004     CASE Maximized!
005     MessageBox("Window Message", "Maximized!")
006     CASE ELSE
007     MessageBox("Window Message", "Unkown Error")
008 END CHOOSE

```

- Listing 17. Using enumerated types .

Standard Types

The other native data types in PowerBuilder are often called the *standard* data types. These are the data representation choices the language offers you of its own accord for storing the data values used in your programs. Each standard data type consist of storage requirements, valid value ranges, and a standard set of operations that can be performed on them. The standard types are used to define your own literals, variables, constants and data members of composites. PowerBuilder provides several standard data types for you to work with, each with its own properties and purposes. In PowerBuilder, these include **Boolean, Real, Double, Integer, Long, String, Blob, Date, Decimal, UnsignedInteger, Character, DateTime, Time,** and **UnsignedLong**, most of which you are probably familiar with. It also includes arrays of any of these, since arrays are really just groups of data items of the same type.

Class Types

In PowerBuilder, as in C++, Smalltalk, Eiffel, and other object-oriented languages, you are not restricted to using only the native data types in your code. You can also create your own data types according to your own needs and specifications. Once defined, you can use these data types in your code just as you would any of the native data types—with some differences. These data types are defined by a class definition and are referred to as objects at runtime. Variables declared of one of these class data types are what I've been referring to as references because the value in the variable is simply an address, or reference, to the location of the object in

memory. The variable does not actually contain the object itself. This is part of the topic of the section on declarations, later in this paper. Powersoft has provided you with a basic set of class data types in its System Class Library, which we'll explore at the end of this paper and then further throughout the book. Examples include Window, CommandButton, DataWindow, Menu and Transaction.

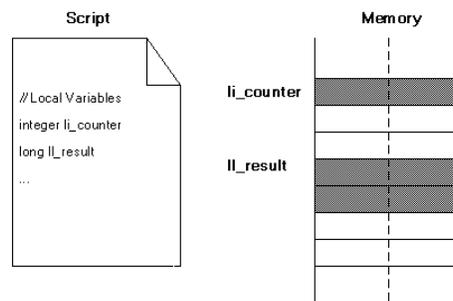
Composite Types

Composite types include structures, arrays, strings and objects. Composite types are constructed from enumerated, standard and class type variables. Objects are the most powerful kind of composite type, and since I'll be focusing on them throughout the rest of the book the discussion here will be limited.

Structures

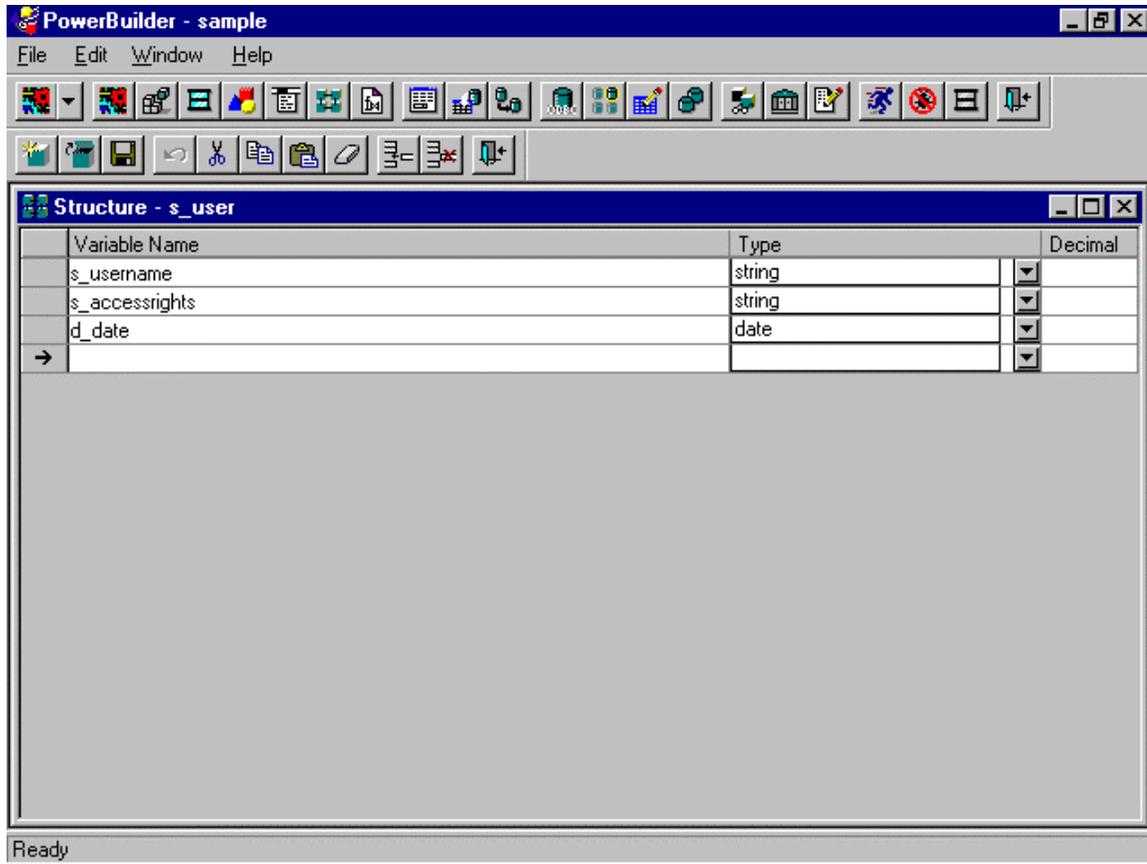
As described in the section on declarations, PowerBuilder allows you to create your own data structures. These are similar to records in Pascal or COBOL. The data items within a structure are often referred to as members or field. Structures give you the flexibility to customize your data representations to fit them to your needs. Using them correctly, though, requires some understanding of how PowerBuilder represents your data in memory.

When you declare a simple data item, such as an Integer to hold a loop index, PowerBuilder allocates the necessary memory to hold that data item. You can count on this space being big enough to hold a value between -32,768 and 32,767. It might be larger, but in the interests of portability, you shouldn't make any assumptions about it. Similarly, when you declare a Long, PowerBuilder sets aside the amount of space in memory that it needs to hold a signed 32-bit whole number—that is, a value between -2,147,483,648 and 2,143,483,647. See Figure 2.



- Figure 2. Statements to declare an integer and a long variable, and the memory chunks that PowerBuilder allocates to hold those variables.

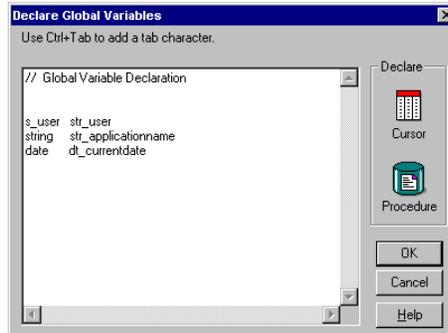
When you declare a data structure, though, things are more complicated. First, you must tell PowerBuilder what your data structure looks like. A structure member can be a variable, enumeration, or composite. As shown in Figure 3, you can use any of the enumerated, standard or class data types as elements in your structure.



• Figure 3. Selecting the types of the data structure members in the structure painter.

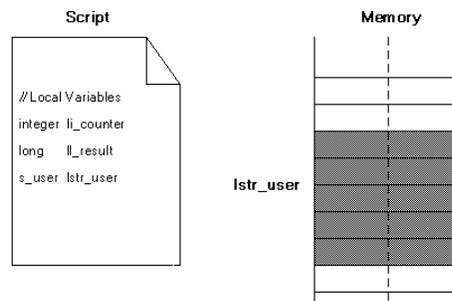
When you have finished defining what members go into your data structure, you leave the Structure Painter. On the way out, PowerBuilder prompts you for a name under which to save your work. The name you give becomes a new type name available in your application.

There is another step. In order to put data in a structure of this shape, you now have to declare a structure having your new type. How you do this depends on where you need it, just as it does for any other variable. If you need to access it from everywhere in your application, you can create a global structure. To do this, go to the **Declare | Global Variables...** menu item from the Window, Menu, or User Object painters. You get a list box like the one in Figure 4.



• Figure 4. Declaring global variables of native data types and of user-defined data types.

As the figure shows, you enter a declaration for the global structure exactly as you would enter a declaration of an Integer or Long. At this time, PowerBuilder allocates the space in memory to hold your structure. See Figure 5.



• Figure 5. Once you have declared the structure, PowerBuilder allocates memory to hold it.

If you need only local access to a structure, you declare it in the necessary script just as you would an Integer or String variable. When the script ends, the structure vanishes, along with all the other local data items in the script. When the script runs again, PowerBuilder creates new instances of all the local variables.

Data structures offer you a convenient way to organize groups of variables that you often need to reference together. You can assign to a single one of their elements using the PowerBuilder dot notation, or to an entire structure, as shown in Listing 5-2.

```
001 str_employee gsCurEmployee
002 str_employee lsTemp

003 gsCurEmployee.LastName=as_NewName
004 lsTemp = gsCurEmployee
```

• Listing 18. Using data structures.

Structures are one of those language elements that Powersoft included in PowerBuilder to support a particular style of programming. Later in this Paper I'll make the case that

NonVisualObjects give you all of the features of structures, plus more, so structures are largely unnecessary in an object-oriented programming style.

Arrays and Strings

An array is a record (or structure) whose members are all of the identical type. Additionally these members are not named as in a structure, but indexed and referred to by a number representing position within the array. Because all of the members are identical, and indexing makes accessing an array very efficient, it is relatively easy to search through arrays and reorganize their order. Suppose you declare an array of Integers with the following statement:

```
001 Integer MonthlyRates[12]
```

- Listing 19. Declaring an array .

Now you have a dozen integers, one right after the other, sitting in memory ready to be filled. Now suppose your script has this bit of code in it:

```
001 RateIndex = 17
.
.
.
002 MonthlyRates[RateIndex] = 500
```

- Listing 20. Using an array index .

The C programmers are probably feeling alarmed at this point. If you created this code fragment in C, you would have just run off the end of an array, and very likely overwritten some other part of data with the assignment. But PowerBuilder handles this differently. If you coded this in PowerScript, you would now have an array of 17 elements, the last of which contains the number 500. This feature offers some clear advantages, for those times when you really don't know at design time how long an array will need to be. But it also has some pitfalls in terms of efficiency and performance.

Strings behave in much the same way and are basically just arrays of characters. The following two variables are identical, except that `lc_string[]` is harder to manipulate than `ls_string`.

```
001 char lc_string[ ] = {'s','t','r','i','n','g'}
002 string ls_string = "string"
```

- Listing 21. Character arrays compared to strings .

PowerBuilder provides the underlying manipulation of character arrays (strings) automatically for you, which makes strings much easier to use than character arrays. For example, if you declare

two strings of fixed length and then concatenate them both into one of them, PowerBuilder will expand the target string to hold the result for you. The limitation here is that strings can hold no more than 32,000 characters.

Attributes (or Properties)

In most of the early programming languages such as Basic and FORTRAN, you could access all of your variables from anywhere in your program, simply by referring to them. In some languages, you could even create new variables simply by using their names—if the compiler didn't recognize an identifier, it assumed it was a reference to a new variable, and assigned it a type based on the first character of the name or the first value assigned to it. Clearly this made perfect typing skills an important asset.

Aside from the problems of inadvertently created variables, universal access to all variables from anywhere in the program generated its own problems. Particularly in very large applications or those maintained by several people, the chances were high that programmers would reuse variables that were already doing something else. This was one of the big motivations for the development of subprograms, modules and subsequently information hiding techniques, as discussed elsewhere. Variables contained within a module, particularly a class module, are generally referred to in object-oriented programming languages as attributes. In its infinite wisdom, Microsoft has chosen to popularize a new term for the same concept - properties. I'll use the more standard term attribute throughout this book, but you can substitute the term property as Powersoft has in PowerBuilder 5.0. In PowerBuilder, you can restrict the visibility of variables declared within modules (attributes or properties) so that only the parts of your application that really need to know about them can access them. This section describes how, once you've declared attributes, you can control the scope and access to those attributes in your PowerBuilder applications.

You must declare your variables before you can access them. This is straightforward and obvious when you are using your basic native data types, but can get a bit more complicated when you begin to create your own data types. This section discusses declaration issues, and explains the distinction between declarations and definitions, a common source of errors.

Declarations

Before you can access a variable, you must declare it. That sounds reasonable enough. In a simple script with a **For** loop, you might define a variable to act as an index with the statement:

```
001 Integer index
```

- Listing 22. Simple variable declaration .

Thereafter, you can use that index variable anywhere in the script.

But when you create your own data types, things get a bit more involved. For example, suppose you were creating a payroll application. Such an application needs to keep track of information about employees, including the employee ID number, name, address, and birthdate of each employee. Rather than creating separate variables for each of these items, it makes more sense to create a structure—a grouping of these different but related data items that you can then use to refer to them all together. To do this, you would create a structure, perhaps called `str_employee` (there is a button on the Power Bar to create structures, in case you've never done

it). Within that structure, you would declare strings for the ID, name, and address, and a date item for the employee's birthdate. This process is known as *defining* the structure. Once you have done that, you can use the structure identifier (`str_employee`) in the same way you use the other data type identifiers such as **Boolean** or **Integer**.

It's important to keep in mind the difference between definitions and declarations. The definition of a data item is not a declaration of it. The *definition* is simply where you describe what the item looks like. In the Structure Painter, you define the layout of your structure. When you want to use a structure having that layout, you *declare* a variable to hold an address to an instance of it. Until you do so, you can't use the structure. This is parallel to how you use native data types; until you declare a `String ls_name`, you can't use it anywhere. But in that case, the data type is already declared for you. When you are creating your own data types, it's easy to forget that you have to go through two steps, especially when you're new to the process. As you get accustomed to creating your own data types, it becomes second nature, but until then, it is the source of annoying mistakes for most beginners. Another source of confusion stems from the fact that, though they look a bit alike (especially if you come from a C++ background), structures do not necessarily behave exactly like instances of classes.

The terminology certainly contributes to the confusion. To begin with, the two words *declare* and *define* are quite similar. Furthermore, the usage has wrinkles of its own. For instance, when you are in the act of *defining* a structure, you *declare* its elements. Listing 1 shows a group of declarations.

```
001 Integer Index
002 String Name, Address, State = "OH"
003 str_employee CurEmployee
```

- Listing 23. Sample variable declarations. The third declaration declares an instance of a previously defined structure.

Scope

The scope of your variables reflects how far they can be seen within your application. The earliest languages had only one type of scope: global. PowerBuilder offers you a choice of four: local, shared, global, and instance. Additionally, when PowerBuilder searches for a variable name it searches in exactly that order.

Global Scope

Global variables, for instance, are the most visible. They can be seen, and therefore used and changed, in any script in your application. No matter where you are coding in your application, if you have any global variables, you can use them there. Global variables can be constants, variables, enumerations, or composites. They can contain data values, or references to objects. In an object-oriented programming style the only use for global variables is for references to objects.

PowerBuilder offers five default global object references, for the Transaction object (`SQLCA` is the variable name), the `DynamicDescriptionArea` (`SQLCA` is the variable name), the `DynamicStagingArea` (`SQLSA` is the variable name), the Error object (`ERROR` is the variable name), and the Message object (`MESSAGE` is the variable name). These are appropriately

global to the application, because PowerBuilder needs to make them available to all of your application in a consistent way, but it can't predict where or how you will need them. If you need to create others global object references, you define them in any of the class painters (Window, Menu, UserObject, Structure, Application), then declare them from the **Declare | Global Variables** menu item within any of the painters that offer it.

As you have probably heard a hundred times, it's highly preferable to avoid global variables and functions. The only exception is global object references. Otherwise, they decrease reusability and increase the tightness of the coupling between elements of your application, and they reside in memory during the entire time your application runs, rather than only when they're needed.

If you use sound analysis and design techniques, you will very never require global data variables. To get rid of the ones you already have, here are some useful tips. If you already have global structures in your application, it's an easy matter to create a custom class in the user object painter and move the attributes of the structure to the attributes of the custom class. Now you can inherit from the new custom class, and then extend or override it as necessary, instead of having to create a new structure to make changes.

If you have global variables, look for ways you can make them attributes of a custom class instead—without making overly contrived connections between them.

Instance Scope

An instance variable is associated with a particular object, and is only accessible from within the event scripts and functions that go with that object unless you allow otherwise. They are created when the object is created and destroyed when the object is destroyed. If you have more than one instance of a class, you will also have separate copies of the instance variables for each instance of the class.

You create instance variables from the **Declare | Instance Variables...** menu item from the painter of the object with which you want to associate them—the Menu, Window, or User Object painters.

Shared Scope

Like their cousins the instance variables, shared variables are associated with objects. The difference is that while an instance variable is associated with a particular instance of an object, shared variables persist across all instances of that object. They are called class variables in Smalltalk and other object-oriented languages. When you instantiate a class in your application that has a shared variable, that variable gets created, and initialized if you so specify. When another instance of that same object gets created, though, PowerBuilder does not create a new instance of the shared variable. Both instances of the object reference, and modify, the same copy.

To create shared variables, select the **Declare | Shared Variables...** menu item from the painters of those objects that support them (Window, Menu, UserObject and Application).

Local Scope

This is the most restrictive scope. Variables declared inside a script have local scope, which means that they can only be seen and used within that script. When the script terminates, they cease to exist. They are appropriate for looping indexes, placeholders, and other temporary elements that do not need to persist beyond the execution of the current script.

Access Rights

Access rights are sometimes confused with scope, but is actually a means of qualifying scope for instance variables. The careful application of access rights is an important tool for an object oriented implementation of your design. You apply access rights to both the variables and functions of an object (its attributes and methods).

Public

Public access is the widest access right, as its name suggests. You'll seldom use public access for the attributes of an object; this would permit any variable or script within its scope access and change the values of those attributes. Instead, a more common object-oriented solution is to provide public scope methods (called public functions in PowerBuilder) that provide services to let other objects reference and change the attributes. This promotes encapsulation: if you need to change the structure of the data within the object, you can do it with less chance of affecting other parts of your application, as long as the interface to the service functions remains the same.

Public access is the default for your methods and objects. In other words, if you don't explicitly define them to be private or protected, PowerBuilder makes them public. Public access rights for instance variables violates the principle of encapsulation and should never be allowed. I'll expand on this guideline later in the section of the programming styles supported in PowerBuilder.

Private

This is the most restrictive access type. Private functions in your objects are those that only the object itself ever needs to execute. If no other object will ever need to access the function (that is, if no other object ever needs to send this type of message to this object), then make the function private.

Many of the attributes in your objects will likely be private, too, except those that need to also be available to descendants. For this purpose, PowerBuilder offers the Protected access.

Protected

Variables and functions with protected access are only visible to the object itself and its descendants, and so is only meaningful for objects from which other objects are inherited. Any attribute or method that the object should pass on to its descendants should therefore be declared Protected. Good object-oriented designs make extensive use of protected access and should generally be your default access rights for instance variables.

Messages and Methods

Having made a fairly clear-cut distinction between functions and events earlier, I must now point out that the picture is not so clear-cut in PowerBuilder 5.0. It is now possible to call events—to synchronously execute the code associated with an event - as well as pass it parameters directly and obtain return values. You can also now post functions—direct the system to asynchronously execute a function - as well as postpone type checking until runtime. For all outward appearances, there is now no distinction between functions and events in PowerBuilder 5.0. This blurring of functions and events is achieved largely with the introduction of a trio of new keyword combinations - TRIGGER/POST, STATIC/DYNAMIC, and FUNCTION/EVENT, which we'll cover shortly. Other considerations, however, remain.

Functions and events permit you to implement the response to messages sent between your classes that you discovered during analysis and design. This is an important distinction for object-oriented programming - the difference between the message and the response. The message is the function or event call. The response to the message is the series of instructions executed as a result of receiving a particular event or function call. The response is either the text of a script in PowerBuilder, the text of a stored procedure in a RDBMS like Oracle or Sybase, or the text of an external function (written in C++ or another language callable from PowerBuilder) and is more formally called a method in object-oriented design and programming. Throughout the rest of the book I'll refer to the function or event call as a message, and the script or program that is executed in response to that message as a method. This section describes the issues involved in sending messages and the various characteristics to consider. It will hopefully help you make choices between functions and events as the form of the message. The considerations for messages are the direction, access rights, name resolution, timing, and parameters of the message. The primary considerations for methods are computational intensity and physical location

Message Direction

If your design calls for bi-directional messages, you need a mechanism that can carry information on both the send and the return. In earlier versions of PowerBuilder only a function would suffice, because it was the only mechanism that provided for the formal definition of parameters and return values. If your messages required only a single direction (i.e. a "send" or notification), a subroutine would have been appropriate if parameters were required. An event would have been appropriate where no data needed to be sent or returned with the message, for example in a clicked event where the target object only needs to know that it has been clicked.

PowerBuilder 5.0 has now blurred the distinction between events and message since it allows the definition of parameters and return values for events. The guidelines I outlined earlier in the paper are still the ones I recommend for deciding which to use. Additionally, remember that events are intended to be asynchronous and are posted to the application message queue to be processed when they come to the top of the queue. A critical return value cannot be processed by the calling object if it doesn't know when the event will be processed. In this case, a function is a better choice.

Message Access Rights

Message access rights is similar to the access rights I discussed earlier for instance variables. It refers to the capability of a developer to restrict the visibility of message names outside of a class. PUBLIC access rights makes the message name visible to all other objects.

PROTECTED access rights restrict visibility to objects of the class, and all descendants of the class. PRIVATE access rights restrict visibility of the message name to the object itself. Only object functions and subroutines can be assigned access rights. Events and global functions have public access by default. If you need to restrict access to a message in order to encapsulate an object, you must use an object function to implement the method. This is one of the primary reason for choosing between functions and events in PowerBuilder 5.0. By definition all events become part of an object's public message interface. The key to a good public message interface is that it's easy to understand and use, and offers the minimum number of messages necessary for the users of an object to obtain the services they need.

Message Name Resolution

When an object receives a message it has to resolve the name in order to determine which method to implement in response to the message. There are two searches that have to be completed to make this determination. First, the object must look through its own methods to see if it can directly respond to the message. In PowerBuilder 5.0, like in C++, there may be multiple methods with the same name. The only difference may be the number or type of parameters they accept. The object must search through this list to determine which one matches the name, number of arguments and type of arguments supplied in the message. If it cannot find a match, it must then begin to search through its ancestor's methods to see if it has inherited one that it can use to respond to the message. This is, in effect, a very simple explanation of polymorphism. The response to a message takes on the appropriate form as a result of the object's search for the correct method. The first search I described above is what I'll introduce later as operational polymorphism. The object searches it's own "operations" to find the one that matches the message. The second search I'll describe later as inclusion polymorphism since the object searches through the methods "included" in it from an ancestor. The important distinction to note here is that objects support inclusion polymorphism for events and functions, but only support operational polymorphism for functions. This will affect your choice of events over functions for implementing methods, usually in favor of functions.

The other important aspect of message name resolution to be familiar with is the new syntax for sending messages. As an example consider the traditional PowerBuilder function call in a "Save" commandbutton on a window:

```
001 w_main.wf_save( )
```

- Listing 24. Traditional PowerScript message .

This message name is checked by the compiler and resolved to ensure that the function (wf_save) exists in the target object (w_main). In the next form of the same message the new DYNAMIC and FUNCTION keywords are used, along with the PARENT pronoun just to make the message really generic, to tell PowerBuilder not to resolve the message name until runtime.

```
002 PARENT.DYNAMIC FUNCTION wf_save( )
```

- Listing 25. New PowerScript message syntax for functions .

The result here is that this commandbutton can be developed independently of any window object (i.e. w_main) and this message can be compiled and will work as long as, at runtime, there is a function called wf_save() in the parent of this button. If there is not PowerBuilder will generate an application error at runtime and terminate the application. This raises yet another distinction between events and functions. The following message may accomplish exactly the

same result, but if the event (ue_save) is not found PowerBuilder will not generate an application error and will simply continue the application and nothing will be “saved” because the operation was never found or executed.

```
001 PARENT.DYNAMIC EVENT ue_save ( )
```

- Listing 26. New PowerScript message syntax for events .

Message Timing

The original message sent to w_main in the previous section was a function call and its associated method would be executed synchronously. The calling object would have waited until the function wf_save() was completed before it could continue executing after the message was sent. The original message above is exactly equivalent to:

```
001 w_main.TRIGGER STATIC FUNCTION wf_save( )
```

- Listing 27. Controlling the timing of messages using TRIGGER .

The new TRIGGER keyword indicates that a method should be executed immediately upon receipt of a message, and the caller will wait until control is returned. TRIGGER behaves similarly to the traditional PowerBuilder function TriggerEvent(“eventname”). Alternatively, the POST keyword behaves like the traditional PostEvent(“eventname”) function. It will cause an event, or function now, to be posted to a queue to be executed asynchronously. The caller can then continue processing without waiting for a response.

```
001 w_main.POST STATIC FUNCTION wf_save( )
```

- Listing 28. Controlling the timing of messages using POST .

Message Parameters

A message may contain one or more parameters that are subsequently passed to the method (or subprogram) that’s executed in response to the message. The method will accept and utilize these parameters as arguments in the body of the instructions it is executing. Both events and functions may accept parameters. Parameters are the primary way in which data and information is passed along with a message. Traditionally events have not accepted parameters, and based on the guidelines I’ve discussed earlier this should still be the most common use of events.

Method Computational Intensity

Methods that contain intense computations, like scientific or financial calculations, are good candidates for a lower level language than PowerScript. PowerBuilder allows you to call functions written in external languages (C and C++ mainly) from within PowerBuilder. These external functions are “wrapped” with a PowerScript message so other PowerBuilder objects never realize that the response to a message they’ve sent is actually being processed outside of PowerBuilder. The messages are declared as Local External Functions from within any one of

the PowerBuilder painters. The method is then written and compiled using the external language's development tools (i.e. the Watcom C++ compiler).

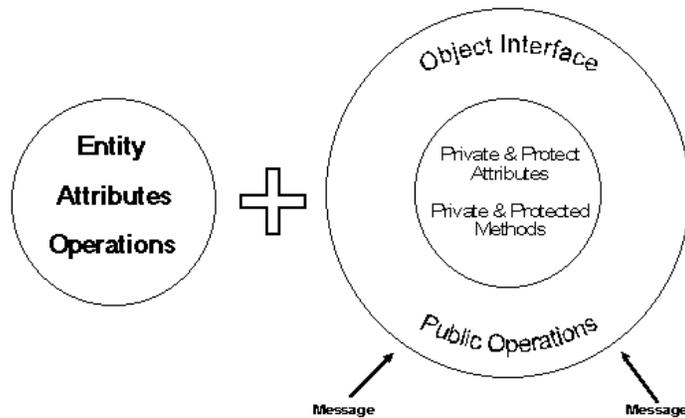
Method Location

Methods that may be shared among more than one application, system or user at the same time may be good candidates for implementation outside of your PowerBuilder application. There are essentially three options for locating methods outside of your application - remote PowerBuilder objects, external objects or functions written in a language like C++ that are accessed via Remote Procedure Calls or Object Request Brokers, stored procedures stored in your relational database management system and executed from with PowerBuilder using the EXEC REMOTE SQL statements. Designing distributed applications using these techniques is a very advanced, and rapidly evolving, type of system development. I'll discuss it briefly later in the book when we discuss the use of "Distributed PowerBuilder".

Encapsulation

Until now we have been discussing many of the lower level language specifics of PowerScript that are present in both procedural and object-oriented languages. These next three sections on encapsulation, inheritance and polymorphism provide the foundation for the object-oriented features of PowerScript.

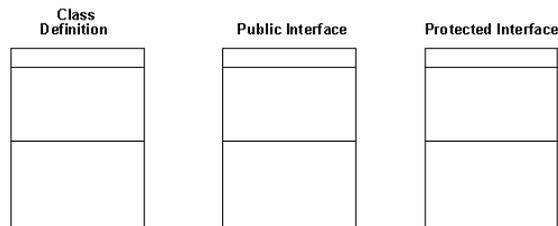
Encapsulation is the foundation of all object-oriented programming. I introduced the term module earlier as a basic concept in all modern programming languages. Encapsulation is the process of creating these modules by packaging, or bundling, data (shared and instance variables) and subprograms (events and functions) with the body of a module (class definition). Once data and subprograms have been bundled into a module, the module must be designed so that the internal details of it are hidden from its users. This prevents them from needing to know about these details and from being affected by changes to them. The method for designing a module like this is to develop a "public interface" as the only way for modules to interact with each other. The public interface is composed of public methods (events and functions) to allow users to request services or data from the module. The result is an encapsulated module, or class.



• Figure 6. Encapsulation includes bundling and information hiding.

The interface to a PowerBuilder class is actually exposed for two different purposes. The first purpose is to provide a way for all other classes in a system to obtain services and data from a class. This is done through public events and functions and is what I've referred to as the public interface. Events are always part of a class' public interface to the rest of the system since they are by definition public. As discussed above, events should be used for sending an object a signal that something has occurred in the system or environment that it should respond to. Public functions are also accessible from outside of a class and should be used as the primary means of gaining access to an objects internal operations that change, compute on or provide data. A public interface can also be inherited by a class from an ancestor, which it then may expand or restrict.

There is also another interface to a class that is available only to its descendants. This can be thought of as the "protected interface" to the class. It is protected from use by objects who are not members of the object's immediate family, or inheritance hierarchy. The protected interface to a class consists of the attributes (instance variables) and methods (functions) that are declared with PROTECTED access rights. These protected members are inherited by descendants and may be used directly, extended or restricted in the descendants. It is extremely important to design the public and protected interfaces separately since they have two different "audiences", but to remember that they will interact with each other. This is one of the reasons that seasoned object-oriented developers often say that analysis and design are much more critical to sound object-oriented programming than they have been for other programming styles.



• Figure 7. Public and protected interfaces to the XXXXX class.

Inheritance

Now that I've introduced the idea of user defined types (classes) I can discuss inheritance in that context, which is really what it provides for. I often find that developers who have spent a great deal of time working with data-oriented approaches to system development have some difficulty with inheritance because they confuse descendant objects with the concept of a subtype. A subtype is simply a restriction on an existing type. This is a valid, and important, distinction but it is not an appropriate criteria for inheritance. All operations that are available on a type are available on all of its subtypes and anywhere the value of that type can be used, the value of the subtype can be used. This is not true of a derived (inherited type). An example of a subtype might be an integer variable called TEMPERATURE that has a valid range of integer values from -273 to 10,000. The TEMPERATURE variable can be used in any expression where an integer type, or any integer subtype, is valid.

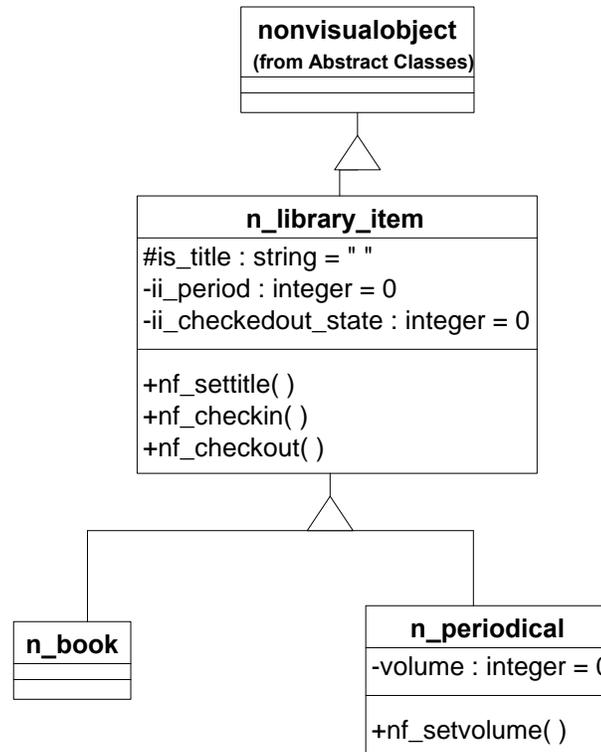
A derived type, on the other hand, cannot necessarily be used in place of the type it was derived from, or other types derived from that ancestor. The derived type inherits the set of all values and the set of all operations from the base type, and can extend or restrict the set of values and

operations. As a result it is not always possible to substitute a derived type for a base type in an expression. The operations being performed may not be valid on the derived type. I'll refer to derived types from now on as descendants.

PowerBuilder makes the use of inheritance extremely easy. All you need to do is click the Inherit button instead of New when opening a painter to create a new Window, Menu, or UserObject class. The reasons for using inheritance, i.e. the design of a class hierarchy, are more complicated. There are essentially two reasons to use inheritance in PowerBuilder. First, if you need to define a new "kind of" something that appears and behaves substantially the same as an existing class, but may be a specialized type of the existing class, you should use inheritance. A book is a "kind of" library item. A periodical is also a "kind of" library item and should be inherited from the library item base class. This is often referred to as implementation inheritance. Book and periodical are concrete implementations of the abstract concept of library item. Second, use inheritance if you need to define a new class that includes the same public interface as an existing class. This is often the case when creating a new "branch" of the class hierarchy. In PowerBuilder, the Transaction and Error classes both are derived from NonVisualObject so they both will include the standard public interface for ClassName(), TypeOf(), TriggerEvent() and PostEvent(). They are not "kinds of" or concrete implementations of NonVisualObjects. Transaction implements a database transaction connection and Error implements an application or system error. They bear no relationship to each other except for the initial public interface they inherit from NonVisualObject. This is often referred to as interface inheritance.

Polymorphism

Briefly, the concept of polymorphism is the characteristic of related objects that may have similar behaviors, but who internally implement those behaviors in many different forms. Another way to say this is that polymorphism is the ability of an instance of a class to assume different types, or the ability to manipulate instances without knowing their types. Remember that the only way to manipulate an object is through its public interface. The ability to do so without knowing the object's exact type means that messages sent to its interface must take on many forms (each different form is actually a different method within the class) depending on the exact type implemented. The result is that polymorphism shifts the responsibility for knowing the correct response to a function call or event trigger from the client to the server. The ultimate result of this is to reduce the overall complexity of your applications when used well!



• Figure 8. Library item class hierarchy.

Static and Dynamic Typing

The implementation and use of polymorphism is much more complicated than the concept itself. In order to illustrate how to design classes that take advantage of polymorphism, you'll need to first understand the ideas of static and dynamic class data type. Remember that a class data type is a class definition (either a PowerBuilder system class, or one you've created yourself) you use to declare a new variable. The new variable then holds the address of the location in memory where an instance of that class can be stored. An example of declaring several class data type variables (reference variables to be discussed in a later publication) is:

```

001 n_library_item    lnv_library_item[ ]
002 n_book           lnv_book1, lnv_book2
003 n_periodical    lnv_periodical1, lnv_periodical2
  
```

• Listing 29. Object declarations .

You've now got a new variable array `lnv_library_item` and its data type is the class `n_library_item`, which is stored in one of your .PBLs. You've also got two other variables `lnv_book1` and `lnv_book2` and they both are of the `n_book` type. These variables now have the static type of `n_library_item` and `n_book`, respectively. The static type is the type the compiler assigns to these variables when they are declared and is what it will use to check any assignments made to these variables. Once you have declared the variables, you need to instantiate some objects:

```
001 Inv_book1 = CREATE n_book
002 Inv_book2 = CREATE n_book
```

• Listing 30. Object creation .

Now you've got two objects in memory and your script can begin to manipulate those objects:

```
001 Inv_book1.nf_settitle("The Design and Evolution of C++")
002 Inv_book2.nf_settitle("Inside the C++ Object Model")
```

• Listing 31. Sending messages to objects .

The compiler checks the data type of `Inv_book1` and `Inv_book2` to be sure they support the `nf_settitle()` operation. Since the `nf_settitle` operation was defined as a public function in `n_library_item`, `n_book` includes it as part of its public interface as well and the compiler is happy because the use of `Inv_book1` and `Inv_book2` was consistent with their static type - `n_book`. But what would happen in the following situation where the `nf_setvolume()` method is only defined in `n_periodical`?

```
001 Inv_library_item [1] = Inv_periodical1
002 Inv_library_item [2] = Inv_periodical2
...
003 FOR itemnum = 1 TO 3
004     Inv_library_item[itemnum].nf_setvolume(0)
005 NEXT
```

• Listing 32. Checking the static type of a message .

The compiler won't allow the message inside the `FOR...NEXT` loop to be sent. The static type of `Inv_library_item` does not support the `nf_setvolume()` operation, even though you know when this script executes that `Inv_library_item[1]` actually holds `Inv_periodical1` (by looking at the array assignments before the loop) and it has the `nf_setvolume()` method in it and should work. This is called the dynamic type of `Inv_library_item[]`. At runtime, you assigned a variable of a slightly different but related type to `Inv_library_item[]` and thus changed its dynamic type as a result of the assignment. Every class variable always has both a static (declared) and dynamic (assigned) type. Many times they are the same, but as in previous example they may not be. The dynamic type may even change more than once during execution of a single script! These are the kinds of things that make sound design such a critical part of object-oriented programming. It's very easy to lose track of the dynamic type if you haven't designed your program carefully.

Inclusion Polymorphism

There are two related types of polymorphism implemented in PowerBuilder. The first, inclusion polymorphism, is based on the idea that a class "includes" all of the attributes and operations of its ancestors, and may override or extend those features (thus "many forms"). This kind of polymorphism has been in PowerBuilder since Version 3.0. When you send a message to a class within a hierarchy, PowerBuilder examines that class, and everything included in it from its ancestors, until it finds the correct method to match with the message it received. The scope of the search is the entire class hierarchy above the class that received the message. This is

considered to be a strongly typed (or “type safe”) language feature since the message is not allowed by the compiler unless a related message exists in the hierarchy.

Operational Polymorphism

The second kind of polymorphism in PowerBuilder is new with Version 5.0. It is formally called operational polymorphism, but is also often referred to as function name overloading or just name overloading. Remember that the basic definition of polymorphism is that an operation on a class can take more than one form. Inclusion polymorphism is the classic implementation of this concept in Eiffel, Smalltalk and C++ and is constrained by the inheritance hierarchy. Operational polymorphism is a language feature that provides alternate method forms within a single class. That is, it is constrained by the class not the hierarchy the class is derived from. Consider the following example:

```
001 lnv_libraryitem[1].nf_checkout(ls_borrowerid, ls_itemid, li_copy)
002 lnv_libraryitem[2].nf_checkout(ls_borrowerid, ls_itemid)
```

- Listing 33. Overloading a message name .

There are two forms of the `nf_checkout()` message. The first form requires a borrower identification, an item identification and a copy number. The second only requires the borrower and item identification. They both have the same message name, `nf_checkout()`, but they take different parameters and respond to the message with slightly different methods. Both forms of the method are defined within the same class - `n_libraryitem`. The best way to remember the difference between inclusion and operational polymorphism is to remember that the scope of resolution for inclusion polymorphism is the class hierarchy and the scope of the resolution for operational polymorphism is a single class.

System Class Library

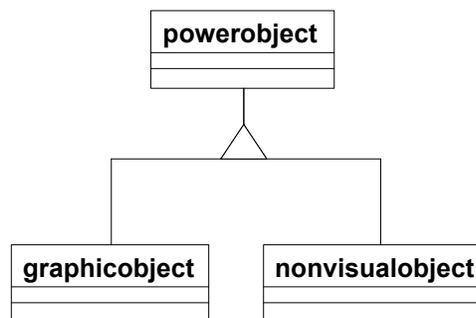
Designing a general library of classes is a difficult undertaking. It's much more difficult than designing an application, since an application must solve a single problem and a general library of classes must be able to solve an entire range of problems. A good library offers one, or a small number, of consistent style that can be applied to that range of problems using the library. This helps make learning the library easier, and makes it clearer how to apply the library in various situations. Learning the PowerBuilder system class library is an essential part of becoming an object-oriented PowerBuilder programmer. It is also an excellent way to learn some of the basic techniques you'll need to create your own class libraries.

Most pure object-oriented languages come with what is called a system class library. It's a library of classes that are used not only by application developers to create new libraries and applications, but internally by the language and development environment themselves. PowerBuilder comes with a well-developed and carefully evolved library of classes, all of which descend from the `PowerObject`. This system class library is designed to support the development of the client portion of a client/server application. That's the “style” of application it supports. It supports two basic variations on that style - Single Document Interface (SDI), and Multiple Document Interface (MDI). With the addition of add-on” libraries from third parties, or by creating your own, you can extend the range of styles to things like Microsoft's new Project,

Workspace and Workbook styles. My recommendation is to learn the PowerBuilder system class library before you begin experimenting with additional styles. You can begin by viewing the entire tree structure of the library (the system class hierarchy) from the object browser, available through the Library Painter. Let me walk you through some of the key characteristics of the PowerBuilder system class library.

First, it's important to recognize that PowerBuilder supports a number of non object-oriented application development styles (distinct from the "application styles" I talked about above). A MDI style application can be created using any one of these development styles - procedural, object-based or object-oriented. We'll focus on the object-oriented approach to development in this book. But one result of this support for the other development styles is that the class library design, a decidedly object-oriented effort, must also address non object-oriented development. So, in the PowerBuilder system classes only the Application, Menu, Window, UserObject, NonVisualUserObject and control classes are true classes, in the sense that they have both attributes and methods you as an object-oriented developer can add to. Many of the other "classes" in the library are there in order to support these other styles of development. One other effect of this support for non object-oriented development is that everything that goes into a PBL, including your functions and structures, is referred to as an "object" by PowerBuilder even though they are actually classes. For non object-oriented development the distinction between class and object is confusing and unnecessary.

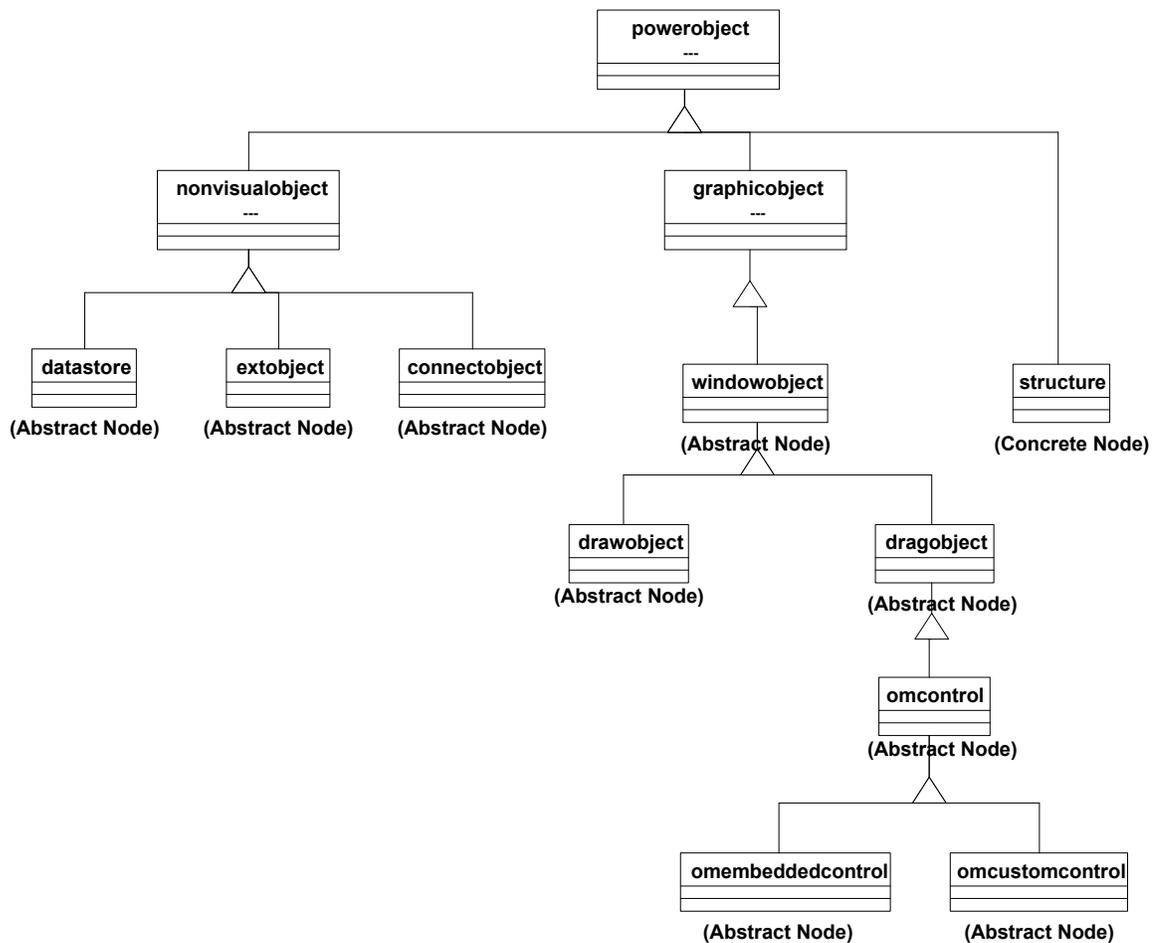
There are essentially 6 types of classes that are used in the PowerBuilder system class library: abstract classes, concrete classes, node classes, adapter classes, proxy classes and representation classes. The system class library begins with PowerObject, an abstract class. Abstract classes are intended to provide a general interface to a range of possible implementations of a common concept. Those implementations tend to be concrete classes that implement that common concept, as I'll discuss next. The methods of an abstract class are often "virtual methods". They don't do anything until they are implemented in a concrete descendant (through inheritance) or in an associated concrete class (through association or aggregation). This separation of the interface in the abstract class from the implementation in a related concrete class allows several implementations of a common concept to exist in a program. Abstract classes typically don't have constructors or attributes, and are not intended to be instantiated.



• Figure 9. Abstract classes in the PowerBuilder system class library.

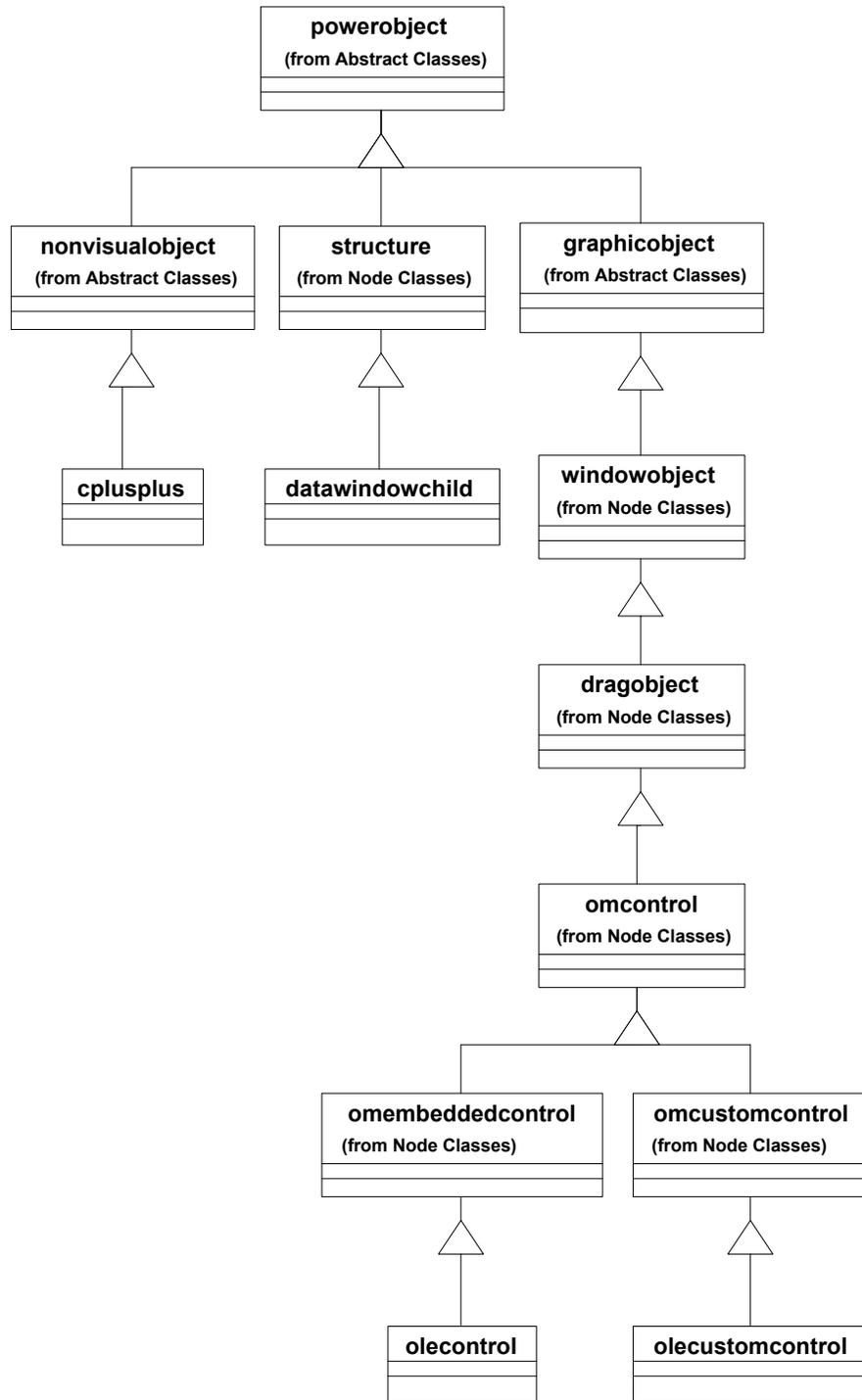
Node classes rely on ancestors for basic services and a public interface, and thus can only be understood in relation to their ancestors. They are used mainly as a base for inheritance and as a common ancestor for a tightly related group of classes. They may have some characteristics in common with abstract classes but they generally provide some additional services for their descendants and may be instantiated, unlike abstract classes. Node classes may also have some characteristics in common with concrete classes if it is not necessary to create

descendants of them to use them. You will often create either abstract or concrete node classes. The real distinction is that a node class, whether abstract or concrete, provides a protected interface to its descendants to allow them to refine or specialize its behavior and those descendants are all closely related to each other and the node class in terms of their primary functionality. Node classes also often have constructors that are essentially for instantiating the node class or its descendants. The `dragobject` and `drawobject` classes in the PowerBuilder system class library are examples of abstract node classes. The `structure` class is an example of a concrete node class.



• Figure 10. Node classes in the PowerBuilder system class library.

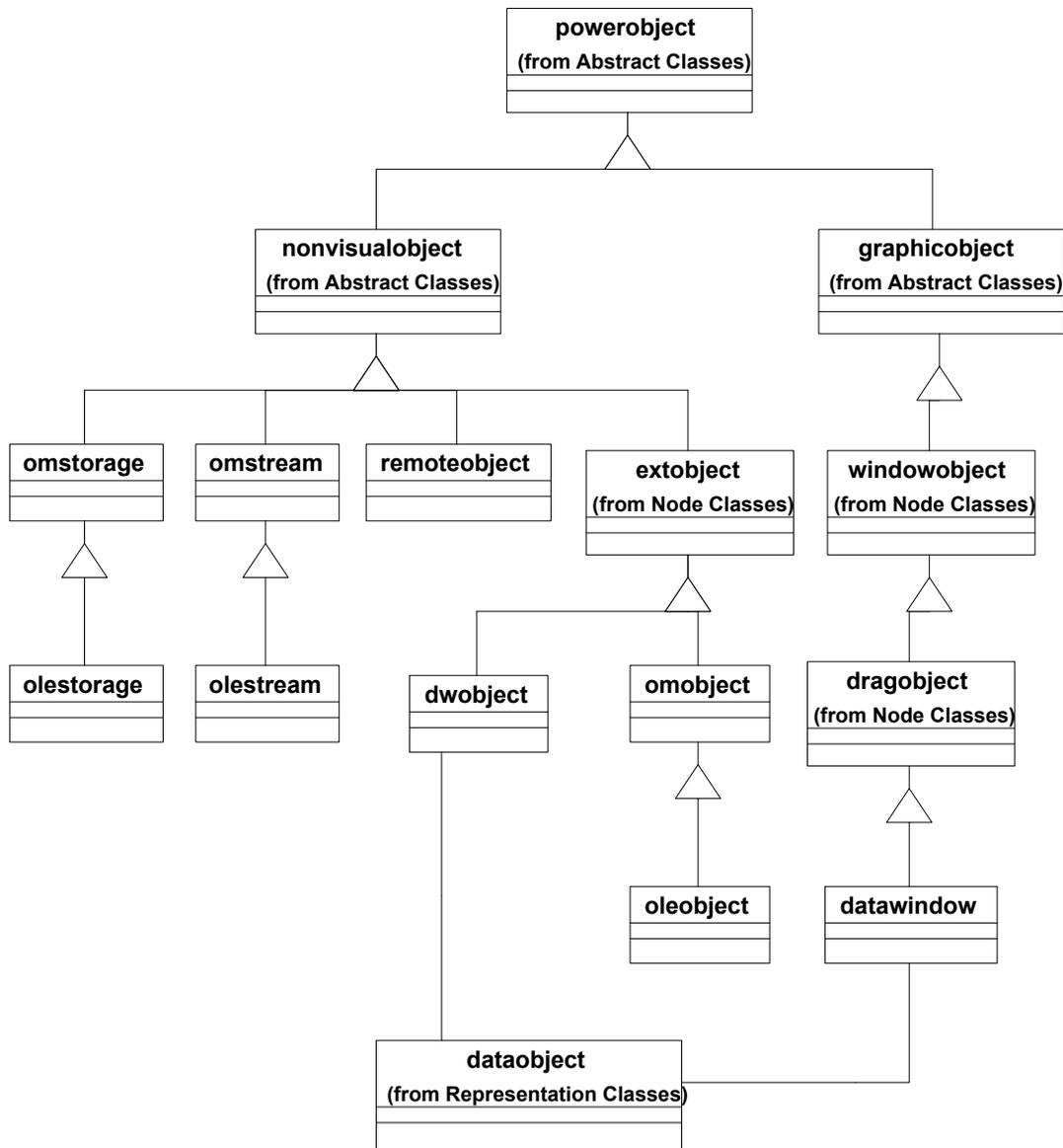
An adapter class (sometimes called an interface class or a wrapper class) doesn't do much more than "adapt" the public interface of an existing class or external application so it can be used more easily by other classes. The adapter really just provides an alternate interface to the existing class or external application. The primary objectives of an adapter class are to make an existing public interface easier to use, insulate the users of a class from changes in its native interface, or to resolve conflicts between the interfaces of two classes. In the case of an external application the adapter class may make it appear to be an object when it really isn't. This is one technique for integrating procedural or legacy applications into object-oriented applications. A good example of an adapter class is the `cplusplus` class in the PowerBuilder system class library. It provides a PowerBuilder interface to C++ objects that reside in external dynamic link libraries (.DLLs).



• Figure 11. Adapter classes in the PowerBuilder system class .

A proxy class (sometimes called a handle class) is one that is used to separate the representation of an object from the interface used to manipulate it. Abstract classes and adapter classes can sometimes function a lot like proxy classes since they also attempt to separate an object's interface from its implementation, but a proxy class is much more loosely coupled to the implementation than either an abstract or adapter class. This looser coupling allows for different representations with different memory and storage requirements to be

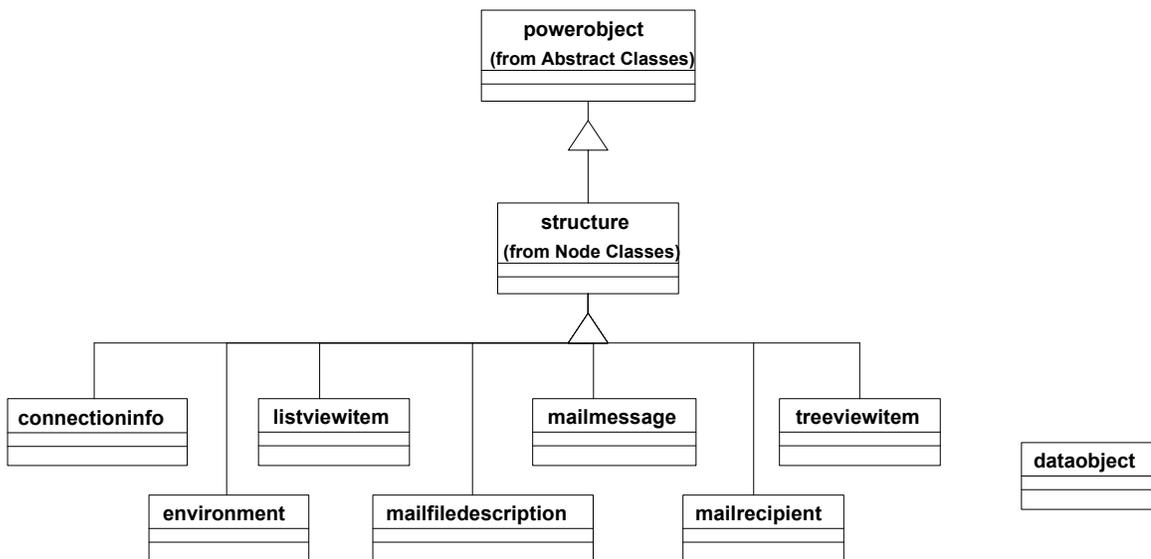
manipulated more easily through the same proxy. The proxy contains the methods and a reference to the representation of the object. The representation contains all of the object's data. Additionally, in distributed applications a proxy represents the interface to an object whose entire physical representation resides on another computer, methods and all. The proxy is essentially a "shadow". The remoteobject is a good example of a proxy class in the PowerBuilder system class library. A DataWindow control class is also an example of a proxy class, as well as being a concrete control class.



• Figure 12. Proxy classes in the PowerBuilder system class library .

A representation class is what is referenced by a proxy. For example, a DataWindow control references a DataWindow object, where all of the data retrieved from a database server actually

resides. The only way to manipulate the information in a DataWindow object is to use the proxy - the DataWindow control.



• Figure 13. Representation classes in the PowerBuilder system class .

Finally, concrete classes are meant to represent some real programming construct you'll need, and they include all of the attributes and methods you would need to create and use objects of that class in a program. There is often a close correlation between its public interface (to classes outside of its family) and its implementation (use of features within itself, and inherited from ancestors). This means that the concrete class provides a public interface to all of the features it contains, and those are all of the features you should ever need in order to use it. Additionally, a concrete class should be relatively independent and usable by itself. The commandbutton class in the PowerBuilder system class library is an example of a concrete class. Most of the classes in the PowerBuilder system class library are concrete classes. This is not normally the case in a system class library where most classes are abstract, nodes, adapters or representations. Since PowerBuilder is intended for use by non object-oriented programmers the system class library is necessarily made up of a large number of concrete classes, which are "filled in" with default attribute values and method implementations using one of the painters. This hides the use of the system class library from developers who are not familiar with object-oriented programming.

Programming Style and Design Considerations

The basic principles of object-oriented programming are not new concepts. Good system designers and developers have always tried to make programs as modular as possible, hide implementation details and reuse common code whenever possible. Object-oriented languages like PowerScript just provide more support for these principles than procedural languages, thus making it easier to do. But just like programming in procedural languages, the time pressure and constraints of real projects can cause some of these principles to be compromised. Don't let it happen to you! In this last section I'll give you some guidelines on choosing a programming style within PowerBuilder, and then some design and coding guidelines for using an object-oriented style in PowerBuilder.

Programming Styles Supported by PowerBuilder

Languages like C++ and PowerBuilder's PowerScript are often referred to as "hybrid languages" because they offer multiple programming styles, or programming paradigms, to their users. Both C++ and PowerScript offer procedural, object-based and object-oriented language features so that programmers can choose the style that most suits their needs and experience level. There are several results that I've observed from this flexibility, some of which are good and some of which are not so good. First, programmers who are familiar with a procedural style of programming can learn those features of PowerScript quickly, and become productive with PowerBuilder relatively quickly. Second, this range of styles provides advanced programmers with a wealth of tools and techniques they can use to solve complex problems. Third, the complexity inherent in a tool with this amount of flexibility results in an overall longer learning curve than one that enforces a single approach. In other words, a programmer can begin using PowerBuilder quickly because he can choose a style he is familiar with, but faces a relatively long learning curve to really begin using PowerBuilder well. Fourth, good program and system design is difficult, and as a result absolutely critical. There are so many design and implementation choices possible in the language it takes a great deal of experience to judge their relative merits, and some combinations of features can cause long term flexibility and maintenance problems that a novice or even intermediate programmer could never anticipate.

The difficulties emerge because often novice or intermediate programmers aren't completely aware of the distinctions between the available styles, and mix them inadvertently. A program written in one of the supported styles tends to be well behaved, but mixed paradigms can hold a wealth of surprises and most them aren't welcome surprises! Let's begin this section with a discussion of the programming styles available in PowerBuilder.

Procedural Style

In a procedural programming style data abstractions (variables, structures, etc.) and the operations that perform on them are declared separately. The operations are ideally implemented as modular functions or subroutines that can be reused throughout one or more programs. These operations are largely task oriented and operate on shared, external data. When one of them is invoked, the calling program must pass them the data structures it wants them to operate on and then wait for the result. This programming style is the one most often used with the Structured Analysis/Structured Design (SA/SD) software engineering methodology I introduced earlier. On the plus side, this style is well suited to procedural languages like C, COBOL and FORTRAN that don't provide support for objects. It also typically requires less analysis and design than object-oriented methods. On the minus side, it results in fewer reusable artifacts and more monolithic system architectures than an object oriented style.

PowerBuilder supports this style of development by providing a number of language constructs that are similar to those found in procedural languages. Global functions and variables, structures, default public access to the properties of all PowerBuilder classes, and an extensive library of system functions are some of the most important PowerBuilder features for this style. The use of these features, in combination with the basic painters - window, menu, function, structure, DataWindow, application, project and library - provide developers with a complete development environment for creating client/server applications without an in-depth knowledge of object-oriented design and programming.

Object Based Style

In an object-based style, data abstractions contain declarations of both data members and the operations on those data members. This style adds a more complete focus on encapsulating data and operations within finer grained modules than in a procedural style. The reason that it's object-based, rather than object-oriented, is that the resulting abstract data types are often manipulated directly either through their set of operations or by directly accessing their data attributes. This can tend to compromise the principle of information hiding and make resulting programs more tightly coupled than a fully object-oriented system. Additionally, inheritance and polymorphism are not required. There is more analysis and design required by this approach, and entity based methodologies like Information Modeling and Information Engineering tend to be very helpful. This style tends to support languages like Visual Basic that provide support for objects, but not true inheritance and polymorphism.

PowerBuilder provides support for this programming style since all of the painters create objects, and there is an additional painter - the UserObject Painter - that allows developers to create their own objects. Data and operations can then be combined in these objects and a complete program can be created by linking these objects together. Global variables and global functions can then be avoided using this style. This style is a good stepping stone to object-oriented development since it introduces the UserObject Painter and the concept of user defined objects.

Object-Oriented Style

In an object-oriented style, abstract data types are grouped into families of related types through inheritance. The abstract base class of the family defines a common interface through which you can manipulate all of the members of that family indirectly using references. Inheritance and polymorphism rely on the use of these references and are key elements of this style. It does however require extensive analysis and design using an object-oriented software engineering methodology to be successful. There are two flavors of object-oriented programming.

Strongly Typed Object-Oriented Style

C++ and PowerBuilder are examples of strongly typed object-oriented programming languages. As we discussed earlier, type checking can be done either at compile time or at run time. Compile time type checking is the main feature I'm referring to when I talk about strong typing. Additionally, at run time if an assignment or message is invalid a strongly typed language will halt the application instead of simply providing an error message and continuing execution. Examples of this can be seen when using PowerBuilder's new DYNAMIC feature. Features like this are provided for advanced developers to be able to control the type checking mechanism in the language to some degree, but should be used cautiously. Strong typing tends to result in more efficient and robust applications. This technique is able to catch more errors at compile time and to create more efficient code when compiled since it can search for and evaluate messages and assignments when it compiles them instead of when it runs them. PowerBuilder in general is a strongly typed language.

Weakly Typed Object-Oriented Style

Smalltalk is an example of a weakly typed language. It does not check anything at "compile" time, and trusts that the developer will provide the correct implementations for messages and assignments at run time. If this isn't the case, Smalltalk will notify the user at run time when there

is an error. This approach is excellent for rapid prototyping and for very experience object-oriented developers.

PowerBuilder contains one feature that behaves in a weakly typed manner - events. Events in PowerBuilder may or may not exist at compile time when they are triggered or posted. Additionally, if they don't exist at run time PowerBuilder just continues executing the application without notifying the user. This can be very powerful, but can also result in a great deal of difficulty when trying to track down application bugs. It can also be expensive at run time when PowerBuilder has to search for an event to execute in response to a message...while the user waits!

Difficulties of Mixed Styles

Programming entirely within one of these styles tends to result in well behaved, predictable, and maintainable programs. Mixing the styles, however, may result in a number of surprises. One of the most expensive of which is that a program written in a mixture of styles is very difficult for other developers to understand and maintain. Unless the styles are mixed for explicitly stated reasons and are well documented, it can make tracking down bugs and adding features much more difficult than it should be.

Guidelines for an Object-Oriented Programming Style

Object-oriented design and programming can only be learned from actually doing it. That means trying things, making mistakes, and learning from those mistakes. The first classes you design and build yourself probably won't be that reusable, but they'll help you learn what does and doesn't contribute to reuse. I can help you get started by giving you some general guidelines for object-oriented development. These guidelines should just be a start, and you should add your own to them as you gain experience.

Class Design

A class is essentially a very small, self-contained program. An object-oriented system is made up of hundreds of these small programs interacting with each other. Each one of them should be designed to perform a specific task or service, and nothing more. The key to good class design is encapsulation, and providing an easy to use interface to all of the behaviors "encapsulated" within a class.

Public Interface

Everything in your system should be expressed as part of the public interface to the classes that make up your system. Attributes are not part of a public interface, only messages that can be sent to a class from outside of it are part of its interface. These public interfaces must be intuitive and easy to use, otherwise people won't reuse your classes. All of the messages that make up those interfaces should be closely related to the classes they reside within, should require a small number of parameters, and should be consistent with the other messages provided in a given class' interface.

Attributes (Shared and Instance Variables)

You'll declare most of your class attributes as instance variables in your custom classes. These should have private access unless they are part of the protected interface to descendent classes, in which case they would be declared as protected. If you do neither, the variables will be declared public by default in PowerBuilder and would violate the principle of information hiding. Public instance variables are one of those features in PowerBuilder that make it easier for developers to migrate to object-oriented development, but should be discarded as soon as you get there.

Methods (Object Functions and Events)

A method is better understandable if it is small (less than 100 lines) and coherent (provides a single, specific operation). Its behavior should be closely related to the class it is implemented within—a given using sound analysis techniques practiced by ASF and other reputable object-oriented analysts. A good technique for method names is to use a "verb-object" pattern (for instance, `set_title`). A list of common verbs and common objects used in an application is a valuable tool for ensuring consistent method names. Take a look at the Microsoft Windows API for inspiration. And remember, methods implemented as events are part of the public interface to a class in PowerBuilder.

Hierarchy Design

Inheritance is not the only way to reuse common code in a class. An association or aggregation to another class provides access to all of the behaviors of that class, and is more loosely coupled than being inherited from it. Inheritance should be used to implement a common public interface among a family of classes, and to provide the foundation for overriding, extending or specializing that interface (polymorphism). When you do use inheritance, pay careful attention to the protected attributes and methods of the base classes. These are the interface to descendent classes, and should be designed as carefully as the public interface to classes outside the hierarchy.

Design vs. Implementation

The actual classes, attributes and methods used to implement an application should exactly match those specified in an object model during analysis and design. Reusable classes don't get created "on-the-fly". They take careful planning and design, and often lots of iterations between design and implementation as you experiment with different things. You may find that you need to do things like introduce additional or different names when you implement a class, add additional public methods or protected attributes once you actually try to reuse a class in a situation different than the one it was designed for, and so on. If you find yourself making these kind of changes during implementation, make sure you go back and update them in your object models so that they match in both places. The object models are your blueprints, and should be used to constantly evaluate the state of your system design.

Program Code Conventions and Standards

The goals of adopting a set of coding and naming standards are readability, reliability and reusability. The three R's of coding and naming standards are closely related. If any experienced PowerBuilder developer can easily read your scripts, you achieve all three goals. Naming conventions, indentation, and comments are three of the most effective tools for making scripts

readable. Consistent coding standards and naming conventions increase the reliability of objects by limiting the ability of the language to cause developers problems. You could give a window object and a menu object the same name, for instance, but you probably won't like the runtime result. This section outlines some of the highlights of the Powersoft coding standards. You'll find a complete discussion, with naming convention standards, in Appendix D.

Case

Use upper case for names of standard data types such as INTEGER and STRING and class names such as N_ORDER. Also use upper case for PowerBuilder key words like FOR, NEXT, IF, CONNECT and so on. Names of predefined PowerBuilder objects, functions or expressions should be written with the first letter of each word capitalized (for example: SetTransObject(), Error, or Message). All other identifiers (attributes, function arguments, or local variables) should be written in lower case. These conventions improve the readability of your scripts if you adhere to them. PowerScript itself ignores case and if you use any of the drop down list boxes in the PowerScript editor it will discard the use of case.

Considerations when Naming Variables

Choose names carefully. All names should accurately describe the classes, attributes and methods that they implement, and should be consistent with their real world counterparts as much as possible. Naturally, cryptic abbreviations are best avoided. When you need to create a multiword names, always put the most important word or words first. A complete identifier, or variable, includes information about the scope of the name, the accessibility of the name outside of the class, the type of value the identifier can hold, and some description of what the identifier is.

Summary

The PowerScript language is much more powerful than many developers realize when they first learn PowerBuilder. The painters hide so many of the features of the language that it takes awhile to really explore them. In this paper I've laid out the basic principles of the PowerScript language, and related them to three different programming style - procedural, object-based and object-oriented. The natural progression is for developers to begin using PowerScript as a procedural language and then progress to an object-oriented style. My recommendation is that this is an excellent approach, but you should remember to discard certain language features as you progress to a new style and begin to utilize new features that support that style. For example, global variables and global functions are examples of features in PowerBuilder you won't see in a fully object-oriented application. Polymorphism is an excellent example of a feature that is best attempted only in a fully object-oriented style.