Applied Object-Based Programming with Blitz3D

Frank Taylor

Overview

The Course

This course requires a working knowledge of Blitz3D Custom Types and the Blitz3D Compiler v1.87 or greater. This course consists of eleven lessons, with new lessons to be added as the course evolves. Each Lesson will have Exercises to help solidify the ideas behind the Techniques. E-mail-in your completed Exercises for Grade!!!

The Objective

Instruct fellow Blitzer3D Users on the application of several Object-Based Programming Techniques that will improve organization, readability, reuse, and expandability of code. The course will attempt to match techniques to those found in Object-Oriented Concepts found in Languages such as C++.

Object-Based Programming

Is a limited version of object-oriented programming with no implicit inheritance and polymorphism. The Object-Based techniques described in this course relies on several rules. Its very easy to deviate from the rules and the only way to force compliance is self-discipline. Blitz3D is not an object-oriented programming language. However, by applying the techniques discussed in this course, you will be able to achieve a high level of Object-Based Programming. OBP will greatly improve your ability to write large Applications and Games.

01: Custom Types Basics

"The Custom Type is the basis for Object-Based Programming in Blitz3D."

Custom Types can be used to create a single item or 'collection' of items that contains a set of variables. It's a programming structure used for grouping a set of variables under a single label. This is the basis for Applied Object-Based Programming with Blitz3D.

```
Type item
Field a%
Field b#
Field c$
End Type

myitem.item = new item

myitem\a% = 1
myitem\b# = 2.5
myitem\c$ = "Hello World"
```

It has been suggested to think of Custom Types as a set of records in a Database, hence the keyword 'Field' used to define them. I would encourage Blitz3D Programmers to also think of Custom Types in terms of Objects. Objects in which each variable is a property that defines the object.

When you create an object using the New Keyword in Blitz3D, you are creating an instance of the object.

Exercise:

1. Create a object type named invader. Define 4 properties: id%, x#, y#, state%
2. Create an invader new instance assigning it to myinvader.invader. Assign the values of: 16,32,1 to myinvader's properties: x#, y#, state% respectively.
3. Print all myinvader's properties to screen.
4. Delete myinvader.

02: Custom Types Advanced

"The Custom Type is the most powerful feature in Blitz3D Language."

The Custom Type features Dynamic Memory Allocation, Pointer Referencing , and Built-in Double Linked List Management. In this lesson you will learn various ways to access and assign data to Custom Type Variables, Arrays, and Pointers.

The New keyword creates a instance of a Custom Type object which is automatically inserted at the front of the collection. Unlike Globals, Arrays, and Variables you can create new instances during the program's run-time without reserving the memory. This is known as Dynamic Memory Allocation.

When you declare a variable of a Custom Type to store a new instance (i.e.: `myalien.alien = New alien`), you are in fact creating a Pointer to the object.

Custom Types Double Linked List Management allow you to use following keywords: Insert, Before, After, First, Last to manipulate the order of objects in the collection sequentially.

```
Insert myalien Before Last alien
```

Think Objects!

To enforce the 'object' mindset, I will refer to Custom Types as Objects From this point forward.

Object Definition

```
Type alien
    Field id%
    Field x#
    Field y#
End Type
```

Returning an Object Pointer from a Method

```
Function alienNew.alien()
    this.alien=New alien
    alien_Index=alien_Index+1 ;global counter
    ;default values
    this\id=alien_Index
    this\x#=20.0
    this\y#=10.0
    Return this
End Function

myalien.alien = alienNew()
```

Passing an Object Pointer to a Method

```
Function alienDelete(this.alien)
Delete this
End Function

alienDelete(myalien)
```

Array of Object Pointers

An Array of Object Pointers can be used to randomly access specific object instances.

```
Dim alien_ID.alien(ALIEN_MAXCOUNT%)

alien_ID(1)=alienNew()

alien_ID(1)\x#=1.0

alienposx = alien_ID(1)\x#

myalien.alien = alien_ID(1)

;Alternative Blitz Array
Global alien_ID.alien[ALIEN_MAXCOUNT%]

alien_ID[1]=alienNew()

alien_ID[1]\x#=1.0

alienposx = alien_ID[1]\x#

myalien.alien = alien_ID[1]
```

Array as a Property

Arrays as a Property can be only one dimension and the number of elements must specified with a constant or value, no variables.

```
Type alienMotherShip
    Field alienstackpointer
    Field alienstack[ALIEN_MAXCOUNT%]
End Type

Function alienmothershipChildPush(this.alienmothership,alienchildID)
    this\alienstack[this\alienstackpointer]=alienchildID
    this\alienstackpointer=this\alienstackpointer+1
End Function

Function alienmothershipChildPop(this.alienmothership)
    this\alienstackpointer=this\alienstackpointer-1
    Return this\alienstack[this\alienstackpointer]
End Function
```

Object Pointer as a Property

Object Pointers are null upon creation of the object instance. You have to devise a method of assignment.

```
Type martian
    Field alien.alien
    Field antennas
End Type

martian1.martian = New martian

martian1\alien.alien = alienNew()

martian1\alien\id=1
```

Array of Object Pointers as a Property

```
Type martianMotherShip
    Field alienstackpointer
    Field alienstack.martian[ALIEN_MAXCOUNT%]
End Type

Function martianmothershipChildPush(this.alienmothership,martian.martian)
    this\alienstack[this\alienstackpointer]=martian
    this\alienstackpointer=this\alienstackpointer+1
End Function

Function martianmothershipChildPop.martian(this.alienmothership)
    this\alienstackpointer=this\alienstackpointer-1
    Return this\alienstack[this\alienstackpointer]
End Function
```

Object and Handle Keywords

These two unique Keywords are specifically used with Custom Types. Although these commands are not documented, many Blitz3D Programmers who are aware of them, rely on them to provide random access to individual object instances. I speculate that Blitz3D has its own internal ID Assignment System for objects. The Handle Keyword returns the object's ID. If the ID is assigned to a variable, the Object Keyword will return the Object Pointer from the variable.

```
handlevar% = Handle(myalien.alien)
myalien.alien = Object.alien(handlevar%)
```

Exercise:

1. Using the invader object you created in the Lesson 01, add the following Properties:
    1. Array with 3 Elements: bomb
    2. Object Pointer: parent.invader
    3. Array of Object Pointers with 4 Elements: child.invader
2. Create an Array of Object Pointers with 64 Elements: invader_ID.

03: Label Convention

"***Readable - Modular - Expandable code***"

Applied Object-Based Programming relies on several simple labelling rules to imitate many OOP features such as Namespaces and Encapsulation. In this lesson you will learn how using a Label Convention will increase your programming productivity. Why you may ask? My answer is simple: by following these rules your code become structured and standardized. Others who know these rules will be able to quickly interpret and extend your code.

Labels are made up of the following words: Prefix, Object, BaseObject, Property, Purpose.

Prefix - Optional acronym for the project. It is used for all labels. Ideal for code libraries.
Object - Name of the Object
BaseObject - Name of an Base Object Pointer
Property - Name of an Object's Property
Purpose - A Verb or short phrase that describes the purpose or action of Object's Method or Construct.

The Rules

Below are the specific rules for labelling an Object's Methods and Constructs.

Constants: All Caps. Each word separated with an underscore(_).

```
{PREFIX}_{OBJECT}_{BASEOBJECT}_{PURPOSE}

Example: Const MYGAME_PLAYER_MAXCOUNT%
```

Globals and Arrays: Capitalize the first letter of Object, BaseObject, Property, Purpose. Each word separated with an underscore(_).

```
{Prefix}_{Object}_{BaseObject}_{Property}_{Purpose}

Example: Global MyGame_Player_Active%
Example: Dim MyGame_PlayerID(MYGAME_PLAYER_MAXCOUNT%)
```

Functions (Methods): Capitalize the first letter of Object, BaseObject, Property, Purpose. Only the Prefix is separated with an underscore(_).

```
{Prefix}_{Object}|{BaseObject}{Property}{Purpose}.{ReturnObject}

Example: Function MyGame_playerWeaponAmmunitionLoad(this.player)
Example: Function MyGame_playerWeaponAmmunitionLoadByPower(this.player)
```

Function Parameters and Local Variables: All lower case. No underscore(_).

```
{prefix}{object}{methodparameter}

Example: Function MyGame_playerMovement(this.player, mygameplayerx#, mygameplayery#)
```

Object Pointer by ID: Capitalize the first letter of Object, BaseObject, Property, Purpose. Each word separated with an underscore(_). Uses Global Keyword and '[]'.

```
Global {prefix}{object}_ID.{ReturnObject}[Elements]

{Prefix}{Object}_ID[objectid]=instance

Example: Global MyGame_player_ID.player[MYGAME_PLAYER_MAXCOUNT%]
Pointer Assignment: MyGame_player_ID[this\id]=this
```

Object Key: 'ID' Suffix added to Field label.

```
Field ID%
Field {baseobject}ID%

Example: Field playerWeaponProfileID%
```

Highly Recommended Reading: Writing Readable Code by Brent P. Newhall

Exercise:

1. Using the OBP Label Convention (no prefix required), create a the following Methods and Constructs for the invader object.
    1. A Constant that represents the maximum number of Invaders: INVADER_MAX. Assign the value of 64 to the Constant.
    2. A Global that counts the number of invaders created: invader_Index
    3. A Method: invaderNew (Constructor). The method performs the following tasks:
        1. Excepts no parameters.
        2. Create an invader instance and assigns it to this.invader.
        3. Add 1 to invader_Index.
        4. Assign invader_Index value to this\id property.
        5. Uses this\id property as the Element Value for the Array Object Pointer invader_ID . Assign this.invader to it.
        6. Return this.
2. A Method: invaderDelete (Destructor). The method performs the following tasks:
    1. Excepts the parameter: this.invader
    2. Delete this
    3. Nothing Returned

04: OBP Basics Part I: Objects

"*An Object is a Person, Place, or Thing.*"

You can look around you now and see an endless number of real-world objects: your dog, your desk, your television set, your bicycle. Each with its own set of properties, behaviours, and purpose. In this lesson , you will learn to program objects by visualizing them in the same manner.

Objects can be modelled from the real-world or completely exist only in software. Both have properties and methods. Let's take a look the following examples:

1. A Bicycle is Real-World Object. We could define the properties of a bicycle: handlebar, seat, frame, wheels, pedals, gears, brake. The purpose of a bicycle is to peddle and steer it, travelling from point A to B. From the description of purpose, we can define the methods of our bicycle: pedal, steer, brake.

We can represent our bicycle object in code like so:

```
Type bicycle
    Field handlebar
    Field seat
    Field frame
    Field wheels
    Field pedals
    Field gears
    Field brake
    Field moving
End Type

Function bicyclePedal(this.bicycle,bicyclepedaldirectionspeed)
    ;Purpose: Move bicycle forward or backward
    If this\moving = True Then MoveEntity(this\frame,0,0,
bicyclepedaldirectionspeed)
End Function

Function bicycleSteer(this.bicycle,bicylesteerdirection)
    ;Purpose: Steer bicycle left or right
    TurnEntity(this\frame,0,bicylesteerdirection,0)
End Function

Function bicycleBrake(this.bicycle)
    ;Purpose: Halt bicycle movement
    this\moving = False
End Function
```

2. A LIFO Stack is a Software Object. We could define the properties of a LIFO Stack: array and element pointer. The purpose of a LIFO Stack is push values into it and pop values out of it in a last-in, first-out order. From the description of purpose, we can define the methods of our LIFO Stack: push and pop.

```
Type stack
    Field pointer
    Field array[STACK_MAXELEMENTS%]
End Type


Function stackPush(this.stack,stackvalue)
    this\array[this\pointer]=stackvalue
    this\pointer=this\pointer+1
```

```
End Function


Function stackPop(this.stack)
    this\pointer=this\pointer-1
    Return this\array[this\pointer]
End Function
```

As you can see, if a thing has a purpose, it can be considered an object. In many cases, you will have to define the object's purpose and behaviour in order to define its properties.

Identifying the Objects

The trick to OBP / OOP is identifying the objects. The first step in identifying Objects is to 'discover' the objects within the task, requirement, features list, etc. In designing your application or game, identifying the objects will be seem more like a challenge in the English Language, not a programming one. Consider the following Game description:

"GunVertor is a Multiplayer FPS in which players can assemble a vast assortment of futuristic guns and ammo to eliminate opponents in both indoor and outdoor arenas."

We discover the following objects from the Game's description by parsing out the Nouns:

* Multiplayer (Networking)
* Player
* Gun
* Ammo
* Opponent
* Arena

Now that we have our objects, we discover the properties of these objects by defining the object's purpose and behaviour. What does the object do? How does it do it? As we breakdown the requirements further, we will discover smaller objects.

When identifying objects they should always be labelled with nouns. As we have seen in the FPS Game example, we picked up nouns from the description of the game. Even when you invent object, keep in mind that they should be nouns. Abstract concepts don't qualify as object names. Method names should contain verbs. In any language, actions performed by nouns are specified using verbs (i.e.: Set, Get, Load, Save).

Prefix adjectives when naming inheriting objects. When a class inherits from a base class, the name for the new class can be determined just by prefixing it with the appropriate adjective. For example, classes inheriting from Gun are called GunRay, GunLaser, etc. Following this convention leads to object names that convey information about the object's inheritance.

Do not add suffixes to object names.

Exercise:

1. Use the following description to create objects and define their properties & methods.
    1. "Invaders From Mars" is an Arcade Game in which the player controls movable laser cannon that moves back and forth across the bottom of the screen. Rows and rows of aliens marched back and forth across the screen, slowly advancing down from the top to the bottom of the screen. If any of the aliens successfully lands on the bottom of the screen, the player is destroyed. The player's laser cannon has an unlimited supply of ammunition to shoot at the aliens and destroy them before they hit the bottom of the screen. The aliens can release deadly rays and bombs that the player must dodge or be destroyed upon contact. The player gets 3 cannons. The Game is over upon the last cannon being destroyed. For Bonus Points, the player can shoot down the AlienMotherShip that flies across the top of the screen occasionally.

05: OBP Basics Part II: The this Pointer

"*Many Object-Oriented Languages contain a special pointer that is called this.*"

In this lesson you will learn how to use the mysterious *this* pointer. The *this* pointer is used within an object's Method to represent the *current* object that is being manipulated. In most OOP Languages the *this* pointer is implicit (in other words, you don't have to type it or see it). With OBP in Blitz3D we have to explicitly designate and assign the pointer.

There a few rules we adhere to that make use of the this pointer clear and efficient:

1. All methods that operate on a instance are passed a object of its own as the first parameter using the *this* pointer.

```
Function alienDelete(this.alien)
    Delete this
End Function
```

2. The *this* pointer can be used in a For... Each construct were no *this* pointer is passed and operation on all object instances is performed.

```
Function alienUpdate()
    For this.alien = Each alien
    ;update this
    Next
End Function
```

3. Do not use the *this* pointer outside a object's Method.

Exercise:

1. Which of the following Method Declarations follows AOBP and Label Convention Guidelines:
    1. Function BombDrop(this.alien,bombtype)
    2. Function alienBombDrop(this.alien,alienbombtype)
    3. Function alienDropBomb(a.alien,bomb)
    4. Function alienDropBomb(alienbomb,this.alien)

2. Find 4 errors in the following Method Declaration:

```
Function CollideBomb.this(b.bomb,player)
    b\y=b\y-1
    If b\y = player\y Then
        If b\x > player\x-5 And b\x < player\x+5
            Return True
        EndIf
    EndIf
End Function
```

3. What is the third rule of using the *this* pointer?

06: OBP Basics Part III: Methods

"Object behaviour and purpose is exhibited through Methods."

Methods are Functions that belong to an object. This ownership is enforced by the use of the Labelling Convention and this pointer. In this lesson you will learn how to define a Method. Also I will introduce to you Constructors and Destructors Methods.

Methods are distinct from standard Functions by applying the following rules:

1. Label Convention for Methods: Capitalize the first letter of Object, BaseObject, Property, Purpose. Only the Prefix is separated with an underscore(_).

```
{Prefix}_{Object}|{BaseObject}{Property}{Purpose}.{ReturnObject}

Example: Function MyGame_playerWeaponAmmunitionLoad(this.player)
```

2. A method that operates on a instance is passed a object of its own as the first parameter using the this pointer.

```
Function alienDelete(this.alien)
    Delete this
End Function
```

3. Methods can only return an object of its own, a base object, and a basic datatype (integer,float,string).

Constructor & Destructors

A Constructor is a Method that's called to create a instance of object. Its purpose is to initialize properties and pointers to the object upon creation.

```
Function alienNew.alien()
    this.alien=New alien
    aliens=aliens+1 ;global counter
    ;default values
    this\id=aliens
    this\x#=20.0
    this\y#=10.0
    Return this
End Function

myalien.alien = alienNew()
```

A Destructor is a Method that's called to remove a object instance. Its purpose is to remove the object and its pointers from memory upon deletion.

```
Function alienDelete(this.alien)
    Delete this
End Function

alienDelete(myalien)
```

With Applied Object-Based Programming, all objects provide a Constructor and Destructor.

Exercise:

1. Using the Label Convention, Create two Methods for the invader object:
    1. A Method whose purpose is to add a new instance to a Element of the child.invader[] property using a Constructor. Hint: you may have to add another property to the object to keep count of child.invader[] Elements.
    2. A Method whose purpose is to delete instance from a Element of the child.invader[] property using a Destructor.

07: OBP Advanced Part I: Classes

“Large Objects built from Smaller Objects"

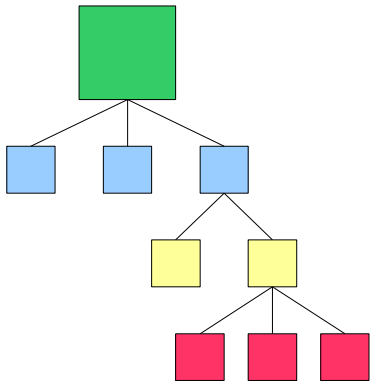
In this lesson, I will explain the purpose of a Class, Hierarchy & Inheritance, and Templates.

Classes

With Applied Object-Based Programming a *Class* is a module or section of code that contains an object and all of its associated methods and constructs (arrays, const, globals). Its completely dependent on the use of the Label Convention.

Hierarchies

A Hierarchy can be visualized as an inverted tree of objects, in which the top object gives its properties to the second set of objects. The second set of objects give their properties to the third, so on and so forth. The mechanism of handing down the properties from object above to the object below is called inheritance.



Object Hierarchies simplify the task of building large programs by building on top or combining basic objects to create more complex objects. Identifying basic objects will not be obvious. In some cases you don't have the basic object, therefore you reverse engineer (breakdown) the complex object to identify the basic objects. This will require you to further analyze the purpose and behaviour objects as you drill down to the basics. Once the basic objects are identified, you can build on *top* of these objects or combine them to create larger objects or derivatives.

For example, in our FPS Game we can have a Class called *Enemy*. The Enemy Class contains a set of properties and methods that are common to all enemies in the game. Such a class is referred to as a *Base Class*. We can create Derivative Enemy Classes that inherit the properties of the Base Class, in addition its own properties and methods when defined.

In Applied Object-Based Programming, inheritance is accomplished with a *Object Pointer as a Property*. Each Object Pointer is initialized with its own Constructor providing an additional set of properties that belong to the object in which the pointer resides. See Lesson 10: Code Wizards.

```
Type enemy
    Field x#
    Field y#
End Type

Type enemytank
```

```
    Field enemy.enemy
    Field cannon
End Type

Type enemytroop
    Field enemy.enemy
    Field gun
End Type
```

Templates

A Template is a object that *shares* it properties with multiple objects that contain an Object Key to link to it. By this usage, Objects do not inherit a Templates properties. Templates are ideal for use as object profiles or where multiple objects can use the property values of single object. Modification to the Template's Properties effect all objects that link to it.

```
;define alienprofile object
Type alienprofile
    Field id%
    Field eyes%
    Field mouths
    Field ears%
    Field limbs%
    Field claws
    Field wings%
    Field fins%
    Field hair%
    Field color%
End Function

;draw alienprofile
Function alienprofileDraw(this.alienprofile)
    ;draw# of this\id%
    ;draw# of this\eyes%
    ;draw# of this\mouths
    ;draw# of this\ears%
    ;draw# of this\limbs%
    ;draw# of this\claws
    ;draw# of this\wings%
    ;draw# of this\fins%
    ;draw# of this\hair%
    ;draw# of this\color%
End Function

;define alien object
Type alien
    Field ID%
    Field alienprofileID%
    Field x#
    Field y#
End Type

;define martian
Type martian
    Field alien.alien
    Field antennas
End Type

;draw aliens based on profile
Function alienDraw(this.alien)
    alienprofileDraw(alienprofile_ID[this\alienprofileID%])
End Function

;create a martian alienprofile template
martianprofile.alienprofile=alienprofileNew()
;set martianprofile properties
martianprofile\eyes%=3
martianprofile\mouths=2
```

```
martianprofile\ears%=4
martianprofile\limbs%=6
martianprofile\claws=0
martianprofile\wings%=0
martianprofile\fins%=2
martianprofile\hair%=2
martianprofile\color%=3

;create some martians and assign them the template id
For martians = 1 to 10
    martian.martian=martianNew()
    martian\alien\alienprofileID%=martianprofile\id%
Next

;draw all martians
for martian.martian = Each martian
    alienDraw(martian\alien)
next
```

Exercise:

1. Answer the follow questions:
    1. Inheritance in OBP is accomplished with what kind of Property?
    2. What kind of Property links a Template to another object?
    3. Using the example *enemy* code above :
        1. What object does the a *enemytank* object inherits the property x# from?
        2. How would you assign a value to *enemytank's* property x#?

    Extra Credit: In your own words, describe a situation in which using a Template is better suited for the job vs Property Inheritance.

08: OBP Advanced Part II: Object Keys

"Tap into Blitz3D Database Functionality."

As mentioned earlier, It has been suggested to think of Custom Types as a set of records in a Database. This analogy can be put to use and serve us well with objects. In this lesson you will learn about Object Keys and how to use them to reference individual object instances using a *Pointer to Object by ID*.

What is a Database Key?

By definition a *Key* is a field that you use to sort records. It can also be called a key field , sort key, index, or key word. For example, if you sort records by age, then the age field is a key. Most database management systems allow you to have more than one key so that you can sort records in different ways. One of the keys is designated the *Primary Key*, and must hold a unique value for each record. A key field that identifies records by primary key in a different table is called a *Foreign Key*.

```
Type alien
    Field id% ;Primary Key
    Field parentid% ;Foriegn Key - Parent alien's Primary Key
    Field x#
    Field y#
End Type
```

Object Keys

A Object Key is a Integer Property that stores a Primary Key or Foreign Key. We use Object Keys to sort and link objects together. We conveniently label a Object Key Property with the 'ID%' suffix (*A personal preference adopted from previous Database Building experience.*)

In using Object Keys, we must devise a way to assign IDs and a *Pointer to the Object by ID*. This requires a hand-crafted ID management System. There are many ways to write a ID Management System, here I will discuss two systems I use and a few others I don't.

Manual ID Management

There two types of systems I use: A LIFO Stack and Global Index Counter.

1. A LIFO Stack is ideal for managing IDs for objects that are frequently created and deleted. The IDs are Recycled. Recycling IDs ensures no IDs are wasted.

2. A Global Index Counter is ideal for managing IDs for objects are created once and never deleted. The Counter is simply incremented by 1 upon the creation of a object.

For both ID assignment systems I use an Array of Object Pointers to *Point to Object by ID*. The array serves a pointer to a specific object instance by matching its element value to the object's primary key Value. ID and pointer assignment is handled within the Constructor and Destructor.

```
Global alien_ID.alien[alien_MAX%];Primary Key Object Pointer
Global alien_Index%[alien_MAX%], alien_IndexPointer%;Primary Key ID Management Stack

Function alienStart()
    ;Purpose: Initialize Class
```

```
    ;Parameters: TBD
    ;Return: None
    For alienloop=ALIEN_MAX To 1 Step -1; Initialize Primary Key ID Management Stack
        alienIndexPush(alienloop)
    Next
End Function

Function alienNew.alien()
    ;Purpose: CONSTRUCTOR - Creates alien Instance; Sets Default Property Values
    ;Parameters: TBD
    ;Return: alien Instance
    this.alien=New alien
    this\id%=alienIndexPop()
    alien_ID[this\id%]=this
    Return this
End Function

Function alienDelete(this.alien)
    ;Purpose: DESTRUCTOR - Removes alien Instance
    ;Parameters: alien Instance
    ;Return: None
    alien_ID[this\id]=Null
    alienIndexPush(this\id%)
    Delete this
End Function

Function alienIndexPush(alienid%)
    ;Purpose: Pushes ID into Primary Key ID Management Stack
    ;Parameters: alienid=Object's ID
    ;Return: None
    alien_Index[alien_IndexPointer]=alienid%
    alien_IndexPointer=alien_IndexPointer+1
End Function

Function alienIndexPop()
    ;Purpose: Pops ID out of Primary Key ID Management Stack
    ;Parameters: None
    ;Return: Object ID%
    alien_IndexPointer=alien_IndexPointer-1
    Return alien_Index[alien_IndexPointer]
End Function
```

I use a *Global* Keyword to declare the Array of Object Pointers with [*Elements*], instead of *Dim* array(*Elements*). This is a Label Convention I use to separate the operation of these arrays from normal arrays. This type of array can be only one dimension and the number of elements must specified with a constant or value, no variables.

Linking Objects by ID

Object ID and *Pointer to Object by ID* are assigned within the Constructor and removed by the Destructor. Its relatively simple to Link Object by Objects Keys.

```
mother.alien=alienNew()
baby.alien=alienNew()
baby\parentid=mother\id
mother.alien=alien_ID[baby\parentid%]

mother\x#=-1.0
alien_ID[baby\parentid%]\x#=-1.0
```

Blitz ID Management

One might consider using Object and Handle commands. Using these commands eliminate the need for manually generating Object IDs and the use of Pointers to Objects by ID. Blitz3D internally generates a unique ID for all objects created. The Handle Command can be used to

assign the ID to the Object Key, and Object Command can be used to retrieve the object from the Object Key. These commands assign and retrieve objects an unlimited number of objects, unlike using an Array of Object Pointers in the hand-crafted ID Management System.

```
Function alienNew.alien()
    ;Purpose: CONSTRUCTOR - Creates alien Instance; Sets Default Property Values
    ;Parameters: TBD
    ;Return: alien Instance
    this.alien=New alien
    this\id%=Handle(this)
    Return this
End Function

Function alienDelete(this.alien)
    ;Purpose: DESTRUCTOR - Removes alien Instance
    ;Parameters: alien Instance
    ;Return: None
    Delete this
End Function
```

The Object Command is use to access the Properties

```
mother.alien=alienNew()
baby.alien=alienNew()
baby\parentid=mother\id
mother.alien=Object.alien(baby\parentid%)
mother\x#=-1.0
```

I prefer not to use Object and Handle commands because: 1) the commands are not officially documented Blitz3D commands; 2) Blitz3D generates IDs only at runtime. The Object ID Counter increments by one upon the creation of any object. So, the FIRST alien object created will be assigned an ID of 14, if it is the 14 object created. It will not be assigned 1 because its the first alien object created. This method of ID assignment can become cumbersome to manage in a Editing Application were object data is edited, saved, and loaded from external files.

When to use Object Pointer or Object Key as a Property?

This question will arise when developing your Objects. There are two guidelines to help make the decision:

1. If the base object has a Primary Key then make the Property a Object Key. If the base object does not have a Primary Key, use a Object Pointer.

2. If the base object will provide property inheritance use a Object Pointer. If the base object is a Template use a Object Key.

Exercise:

1. When using a hand-crafted Object ID management, what two object methods handle ID and Pointer to Object by ID assignment?
2. What programming construct is used as Pointer to Object by ID. What does Pointer to Object by ID point to?

09: OOP Concepts: Encapsulation & Polymorphism

"*Demystifying OOP*"

In this lesson I will make my final stand on demystifying the OOP Concepts: Encapsulation & Polymorphism. In this lesson, you will learn how to provide a limited amount of these two features by how you define a Function/Methods Parameter set.

Encapsulation

Encapsulation is protecting an object's properties from being manipulated outside one of its own methods. Using the Label Convention and OBP Rules provide a great amount of Encapsulation. However, lets consider Class Globals declared with the Global keyword. Globals are accessible to all code in the program. This may not be desirable. To protect Class Globals from outside methods and constructs, we could create a object named objectGlobal in which its properties serve as Globals for the class.

We can also use Class Global Parameters based on Local Variable Scope in which Local variables can only be used within the function they are created in. The Global keyword is not used. Instead, the Class Global Parameter only exists as a parameter defined in all the object methods that use it. The Class Global Parameter is strictly passed between only a object's methods, completely isolating it from outside methods and constructs. Although you can have up to 255 parameters in a function, using a large number Class Global Parameters is not practical.

Polymorphism

Polymorphism is a OOP Language's ability to process objects differently depending on their datatype. Its accomplished by Function Overloading, which is the ability to use a single Function label for multiple purposes. The closest feature we have to Function Overloading in Blitz3D is the ability to assign default values to function parameters. I refer to these as Optional Parameters. By using Optional Parameters we either override the default value by passing an argument, Or omit the argument from the Function call.

Placing Function parameters in a specific Left-to-Right order is key to efficiently using default values to imitate Overloading. Only basic datatypes (integer, floats, strings) can be assigned default values. Object Pointers cannot be assigned default values. Object Pointers are placed on the LEFT of Regular Parameters. Regular Parameters are placed on the LEFT of Optional Parameters. We maximize efficiency, by ordering Optional Parameters based on how often their default values are overridden. Optional Parameters that are most often overridden are placed to the LEFT of Optional Parameters that are least often overridden.

1. *this* pointer - first parameter
2. Object Pointer
3. Regular Parameter
4. Optional Parameter (most often overridden)
5. Optional Parameter (least often overridden)

```
Function objectPurpose(this.object, object pointers, regular parameters, optional
parameters)
```

Exercise:

1. Create invaderPostionToCamera Method using the following parameter list:

   ```
   invaderrelative%=False (least often overridden)
   this.invader
   invaderx#
   camera.camera
   invadery#
   invaderz#=0.0 (most often overridden)
   ```

   Order the parameters Left-to-Right based on their Default Values.

2. Create Class Global Parameter labelled: invaderGlobal%. Apply the Class Global Parameter to all invader methods.

10: Code Wizards

"*Code Wizards work Magic!*"

Introducing the TypeWriter IV Code Wizard. This Utility was specifically designed based on the Rules of Applied Object-Based Programming with Blitz3D.

This Utility will parse a *.object definition file to generate a *.class file that contains a set of common object methods. Class includes:

1. *object*Start: Initializes Class
2. *object*Stop: Shutsdown Class
3. *object*New: (Constructor) Creates an Instance; Initializes Default Property Values; Assigns Primary Key Object Pointer
4. *object*Delete: (Destructor) Deletes an Instance.
5. *object*Write: Writes Property Values to a specified file.
6. *object*Read: Creates Instances; Reads and assigns Property Values from a specified file.
7. *object*Save: Saves all Instances Property Values to a specified file.
8. *object*Open: Loads all Instances from a specified file into Instances.

It's designed to work with all Object Property declarations as explained in the Lesson 02: Advance Custom Types. The utility will also generate needed constructs and methods for Managing Object Key Management and Pointer To Object ID Assignment.

To use it:

1. Unzip typewriter.zip into the Typewriter4 Folder.
2. Load and Run demo.bb in Blitz3D IDE.

The *.object File

The *.object File is a definition of the object. TypeWriter's Parser is very limited. It reads the *.object File line by line (no tabs allowed). It recognizes only the following keywords:

```
type {object name}
field {property label}
max={maximum number of objects in collection; an integer value}
recycle={Primary Key Managment; 0=Counter, 1=Stack}
prefix={Project Prefix Acronym}
```

See the example on the next page...

```
type obj
    field id%
    field a%
    field b#
    field c$
    field d.subobj
    field e.subobj[2]
end type
max=4
recycle=1
```

Typewriter4 requires that each field label to be followed by its datatype extension: %, #, $, .object

Exercise:

1. Create a *.object file of your choice. Modify the demo.bb to execute typewriter v4 on your *.object file.

Appendix

Object-Based Programming Rules

1. Use the Label Convention
    1. Const {PREFIX}_{OBJECT}_{PURPOSE}
    2. Global {Prefix}_{Object}_{SubObject}_{Property}_{Purpose}
    3. Dim {Prefix}_{Object}_{SubObject}_{Property}_{Purpose}(Dimensions)
    4. Function {Prefix}_{Object}|{SubObject}{Property}{Purpose}.{Object}
    5. Function Parameters {prefix}{object}{methodparameter}
2. All methods passed parameters should be passed a object of its own as the first parameter using the this pointer.
3. Methods only return an object of its own or subobject type and basic datatypes (integer,float,string).
4. Do not use the *this* pointer outside a object's Method.
5. Assign Default values to Function Parameters starting on the right to the left.

Glossary

A

abstraction

The process of picking out (abstracting) common features of objects or methods and combining them into a single object or method.

argument

A value passed to function/method call parameters list.

array

A programming structure that stores a series of elements of same datatype/custom type under a single label.

B

base object

A object that contains a set of properties and methods that are common to many other objects.

C

class

A module or section of code that contains an object and all of its associated methods and constructs.

construct

A component of a programming language: constant, variable, array, function, keyword, operator, condition, loop.

custom type

A programming structure used for grouping a set of variables under a single label.

D

derivative

An object that inherits the properties of one or more objects.

dynamic memory allocation

Memory captured and released as needed during a program's execution.

E

encapsulation

A code design which protects an object's properties from being manipulated outside one of its own methods.

F

foreign key

A key field that identifies records by primary key in a different table.

function

A named section of a program that performs a specific task.

G

H

I

inheritance

The mechanism of handing down the properties from object to object another.

instance

The creation of a new object.

J

K

key

A field that used to link and sort records.

L

label convention

A set of Labelling Rules that describe how to label Object Methods and Constructs.

M

method

A function that belongs to an object.

N

namespace

Namespaces group a objects and/or methods under a name.

O

object

A self-contained entity with its own properties and methods.

object key

A Integer Property that stores a Primary Key or Foreign Key within an Object. The property names includes a "ID%" suffix.

object pointer

A variable of object type. Use to reference an object instance.

optional parameter

A parameter assigned a default value.

overload

The use a single Function label for multiple purposes.

P

parameter

A variable specified in a Function/Method Label definition which are passed arguments when the function is called.

Primary key

A key that holds a unique ID for each record.

property

A variable label that represents an attribute of a object.

Q

R

S

stack

A data structure in which items are removed in the reverse order from that in which they are added, so the most recently added item is the first one removed. This is also called last-in, first-out (LIFO).

T

U

V

W

X

Y

Z