SimpleOOP – Opensource OOP Plugin

Einführung

SimpleOOP bringt PureBasic eine einfache OOP Unterstützung ohne komplizierten Syntax, mit besonderem Wert auf einen sauberen und simplen Code.

SimpleOOP ist **Opensource** (Siehe License.txt) damit es für zukünftige PB Versionen keine Probleme gibt und man keine Angst haben muss, dass einmal das Plugin nach einem PureBasic update nicht mehr funktioniert.

Um SimpleOOP zu installieren, starten Sie einfach den "SimpleOOP Installer.exe". Nach der Installation können Sie sofort losarbeiten, der Installer erledigt alles automatisch.

Für den ersten Einstieg lesen Sie sich am besten diese Anleitung durch und schauen Sie sich auch den Examples Ordner an, wo nochmal alle Befehle anhand von Beispielen erklärt werden.

mfg
Sirhc.ITI

Manuelle Installation – PureBasic IDE

- im Verzeichnis "PureBasic\Compilers\" die Datei "PBDebugger.exe" in "_PBDebugger.exe" umbenennen und anschließend die beiliegende "PBDebugger.exe" in das Verzeichnis einfügen

- die Datei "SimpleOOP.res" in das Verzeichnis "PureBasic\Residents\" kopieren

- "SimpleOOP.exe" an eine geeignete Stelle kopieren und dann die nachfolgenden Einträge hierfür in der IDE erstellen

Werkzeuge konfigurieren: (SimpleOOP.exe)

1.
- Name: SimpleOOP Debug Start
- Argumente: DebugStart "%CompileFile" "%CompileFile" "%Path" Format
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Vor dem Kompilieren/Starten
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

2.
- Name: SimpleOOP Debug Stop
- Argumente: DebugStop "%CompileFile" "%Executable" "%Path" Format
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Nach dem Kompilieren/Starten
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" deaktiviert

3.
- Name: SimpleOOP Compile Start
- Argumente: CompileStart "%CompileFile" "%CompileFile" "%Path" ""
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Vor dem Erstellen des Executable
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

4.
- Name: SimpleOOP Compile Stop
- Argumente: CompileStop "%CompileFile" "" "%Path" ""
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Nach dem Erstellen des Executable
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

Info

- Sie können das Argument "Format" weglassen, wenn Sie den veränderten Quellcode im eigenständigen Debugger sehen wollen

- Sie können in den IDE Einstellungen unter "Eigene Schlüsselwörter", "Falten" und "Indentation" , die SimpleOOP Schlüsselwörter hinzufügen

- Durch gedrückt halten der "Strg" Taste beim Kompilieren, wird der veränderte Quellcode aufgerufen

Werkzeuge konfigurieren: (SimpleOOP ClassViewer.exe)

1.
- Name: SimpleOOP ClassViewer
- Argumente: "%TempFile”
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Menü oder Tastenkürzel (Strg + 1)
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" deaktiviert

2.
- Name: SimpleOOP ClassViewer AutoCompleteOnBackslash
- Argumente: AutoCompleteOnBackslash
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Editor-Start
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" deaktiviert

Info

- Sie können das Werkzeug " SimpleOOP ClassViewer AutoCompleteOnBackslash " weglassen, wenn Sie keinen automatischen Autocomplete aufruf nach dem drücken von "\" haben wollen

- Sie können in den IDE Einstellungen unter "Eigene Schlüsselwörter", "Falten" und "Indentation" , die SimpleOOP Schlüsselwörter hinzufügen

Manuelle Installation – jaPBe IDE

- im Verzeichnis "PureBasic\Compilers\" die Datei "PBDebugger.exe" in "_PBDebugger.exe" umbenennen und anschließend die beiliegende "PBDebugger.exe" in das Verzeichnis einfügen

- "SimpleOOP.exe" an eine geeignete Stelle kopieren und dann die nachfolgenden Einträge hierfür in der IDE erstellen

Werkzeuge konfigurieren: (SimpleOOP.exe)

1.
- Name: SimpleOOP Debug Start
- Argumente: DebugStart "%CompileFile" "%CompileFile" "%Path" Format
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Vor dem Kompilieren/Starten
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

2.
- Name: SimpleOOP Debug Stop
- Argumente: DebugStop "%CompileFile" "%Executable" "%Path" Format
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Nach dem Kompilieren/Starten
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

3.
- Name: SimpleOOP Compile Start
- Argumente: CompileStart "%CompileFile" "%CompileFile" "%Path" ""
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Vor dem Erstellen des Executable
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

4.
- Name: SimpleOOP Compile Stop
- Argumente: CompileStop "%TempFile" "" "%Path" ""
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Nach dem Erstellen des Executable
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" aktiviert

Werkzeuge konfigurieren: (SimpleOOP ClassViewer.exe)

5. (Werkzeug wird Manuel aus dem Menü Gestartet)
- Name: SimpleOOP ClassViewer
- Argumente: "%TempFile”
- Arbeitsverzeichnis:
- Ereignis zum Ausl. des W.: Menü oder Tastenkürzel (Strg + 1)
- Einstellungen: "Warten bis zum Beenden des Werkzeugs" deaktiviert

Info

- Sie können das Argument "Format" weglassen, wenn Sie den veränderten Quellcode im eigenständigen Debugger sehen wollen

- Sie können in den jaPBe Einstellungen unter "Custom Keywords" und "Indentation/Completion", die SimpleOOP Schlüsselwörter hinzufügen

- Durch gedrückt halten der "Strg" Taste beim Kompilieren, wird der veränderte Quellcode aufgerufen

ClassViewer

Der ClassViewer hat zwei Funktionen. Einmal das Anzeigen aller Klassen im Quellcode inklusive Includes. Und zum anderen bietet er eine Autocomplete Liste um die Klasse des ausgewählten Objects anzuzeigen.

Der ClassViewer kann Manuel über die IDE gestartet werden. Er listet alle im Quellcode vorhandenen Klassen inklusive deren Attribute, Methoden etc. auf.

Befindet sich der Cursor im Quellcode hinter einem Backslash (z.B. "*Obj\", "Parent\", "This\"), so wird nach dem Aufruf des ClassViewers eine Autocomplete liste angezeigt sofern das Objekt eine Klasse besitzt.

Hat man das IDE Tool "SimpleOOP ClassViewer AutoCompleteOnBackslash" aktiviert, so wird nach dem Schreiben eines Backslash automatisch die Autocomplete Liste aufgerufen sofern das Objekt eine Klasse besitzt.

Die Autocomplete Liste zeigt immer nur die Methoden/Attribute an, die an der aktuellen aufruf Position sichtbar sind. d.h. Public und Protected Methoden/Attribute werden automatisch ausgeblendet/angezeigt. Dies funktioniert auch mit verschachtelten Objekten (z.B. "This\obj\obj2\").

Bei geöffneter Autocomplete/Class Liste kann man durch Schreiben des Wortes oder "Pfeil Hoch/Runter" auf der Tastatur den Eintrag auswählen und per doppelklick mit der Maus oder Eingabetaste in die IDE an der aktuellen Position einfügen.

Befindet man sich mit dem Cursor hinter einem "NewObject.Class", so wird wen vorhanden, nach dem aufrufen das ClassViewers die Init Parameter der Klasse im Quellcode eingefügt.

Mit einem Rechtsklick auf eine Methode, Attribut oder Procedure im ClassViewer, kann man zu der jeweiligen Zeile im Quellcode springen.

Ein Rechtsklick auf einen Node im ClassViewer Faltet alle anderen Nodes zusammen und expandiert den aktuellen.

Polymorphie

In SimpleOOP gibt es die Möglichkeit, Methoden zu überschreiben (Attribute können nicht überschrieben werden). Dabei überschreibt die Kindmethode immer die Elternmethode. Beim überschreiben sollte man darauf achten, dass sowohl Parameter und Rückgabetyp der Methoden gleich sind. Es ist zwar möglich dies nicht zu tun, allerdings kann das zu Problemen führen, wenn z.B. eine Elternklasse eine durch die Kindklasse überschriebene Methode aufrufen will, aber sich deren Parameter geändert haben. Ebenfalls kann nur eine Methode eine Elternmethode überschreiben, wenn sie den gleichen Scopetyp hat, d.h. beide Methoden müssen Public oder Protected sein, eine Protected Methode kann keine Public Methode überschreiben und umgekehrt.

Schlüsselwörter

- Singleton
- Class/EndClass
- Abstract
- Private/BeginPrivate/EndPrivate
- Protect/BeginProtect/EndProtect
- Public/BeginPublic/EndPublic
- Method/MethodReturn/EndMethod
- Parent
- This
- NewObject
- FreeObject
- _ (Multiline Unterstützung)

Attribute

Attribute verhalten sich wie Variablen in einer Struktur. Arrays, Maps und Listen werden auch in Klassen unterstützt.

Beispiel:

```
Class MyClass
        Val.l
        A.q
        B.f
        String$
        Map Tools()
        List MyList()
        Array MyArray(5)
EndClass
```

Singleton

- macht die Klasse statisch
- bei jedem NewObject wird das als erstes instanzierte Objekt zurückgegeben
- der Konstruktor Init() wird nur beim ersten NewObject aufgerufen
- FreeObject wird ignoriert, der Destruktor Release() wird nicht aufgerufen

Beispiel:

```
Singleton Class MyClass
    Method Test()
        ; Code
    EndMethod
EndClass
```

Class/EndClass

- wie Strukturen, unterstützt Extends
- Methoden werden direkt in die Klasse geschrieben
- Prototypes können direkt in der Klasse stehen
- Attributen kann ein fester Startwert zugewiesen werden

Beispiel:

```
Class MyClass
    String$
    Val.l = 123
    List MyList()
    Map MyMap()
    Array NewArray(5)

    Public Method Test()
        ; Code
    EndMethod

    Prototype.i Proc(Num.d, *Pointer)

EndClass
```

Abstract

- zum Erstellen abstrakter Methoden
- Abtrakte Methoden müssen durch Kindmethoden überschrieben werden, andernfalls kann die Klasse nicht instanziert werden

Beispiel:

```
Class MyClass
    String$
    Val.l

    Abstract Test()
    Public Abstract DoThat(Val.l, String$)

EndClass
```

Private/BeginPrivate/EndPrivate

- für private Attribute oder Methoden in Klassen
- private Attribute oder Methoden können nur von der eigenen Klasse aufgerufen werden
- standardmäßig sind alle Objekte einer Klasse privat, dieses Schlüsselwort ist imgrunde nur zur besseren Übersicht

Beispiel:

```
Class MyClass

    String$ ; ist auch Private

    BeginPrivate
        Val.l
    EndPivate

    Private Method Test()
        ; Code
    EndMethod

    BeginPrivate
        Method Proc()
            ; Test
        EndMethod
    EndPrivate

EndClass
```

Protect/BeginProtect/EndProtect

- für geschützte Attribute oder Methoden in Klassen
- geschützte Attribute oder Methoden können nur von der eigenen Klasse und deren Kindklassen aufgerufen werden

Beispiel:

```
Class MyClass

    Protect String$

    BeginProtect
        Val.l
    EndProtect

    Protect Method Test()
        ; Code
    EndMethod

    BeginProtect
        Method Proc()
            ; Test
        EndMethod
    EndProtect

EndClass
```

Public/BeginPublic/EndPublic

- für öffentliche Attribute oder Methoden in Klassen
- öffentliche Attribute oder Methoden können von überall aus aufgerufen werden

Beispiel:

```
Class MyClass

    Public String$

    BeginPublic
        Val.l
    EndPublic

    Public Method Test()
        ; Code
    EndMethod

    BeginPublic
        Method Proc()
            ; Test
        EndMethod
    EndPublic

EndClass
```

Method/MethodReturn/EndMethod

- wie Proceduren
- innerhalb kann mit "This" gearbeitet werden
- Methoden werden direkt in die Klasse geschrieben

Beispiel:

```
Method.s Test()
    ; Code
    MethodReturn "Hallo"
EndMethod
```

Parent

- Zum Aufrufen einer Elternmethode in einer Methode
- Beachten Sie: wenn in der Elternmethode weitere Methoden aufgerufen werden, dass deren Parameter, gleich der, in der Kindklasse überschriebene Methoden sein sollten (Siehe Polymorphie)

Beispiel:

```
Class Second Extends First
    Method.s Test()
        Parent\Test() ; Calls Test() in Class First
    EndMethod
EndClass
```

This

- für den Zugriff auf "sich selbst" in Methoden
- alternativ kann auch *This geschrieben werden

Beispiel:

```
Method.s Test()
    Debug This\Val
EndMethod
```

NewObject

- zum Erstellen neuer Objekte
- unterstützt Objekte, Arrays, Listen, Maps…
- objekte müssen immer mit "*" beginnen
- die Konstruktor Methode "Init()" der Klasse wird automatisch aufgerufen, sofern vorhanden
- nach "NewObject" kann man auch Parameter an den Konstruktor "Init()" übergeben
- nach "NewObject" muß die zu erzeugende Klasse angegeben werden

Beispiel:

```
*Object.MyClass = NewObject.MyClass

Dim *Array.MyClass(10)
    For I=0 To 10
        *Array (I) = NewObject.MyClass()
    Next

    NewList *Liste.MyClass()
        AddElement(*Liste())
            *Liste () = NewObject.MyClass

*Object.MyClass = NewObject.MyClass("String", 123)
```

FreeObject

- löscht das Objekt
- die Destruktor Methode "Release()" der Klasse wird automatisch aufgerufen, sofern vorhanden
- setzt das Objekt automatisch auf 0
- beachten Sie: wenn Sie "This" freigeben, dass das Objekt außerhalb nicht automatisch auf 0 gesetzt wird, da der "This"-Zeiger "byVal" übergeben wird

Beispiel:

```
*Object  = FreeObject
*Array(3) = FreeObject
This = FreeObject

ForEach *Liste()
    *Liste () = FreeObject
    DeleteElement(*Liste())
Next
```

_ (Multiline Unterstützung)

- Um einen Befehl auf mehrere Zeilen aufzuteilen
- Ein Befehl kann auf so viele Zeilen wie man will aufgeteilt werden
- Bei Fehlern auf einer aufgeteilten Zeile, markiert der Debugger immer die erste Zeile

Beispiel:

```
OpenWindow(0, 0, 0, 640, 480, _
                "PureBasic Window", _
                #PB_Window_SystemMenu |  _
                #PB_Window_MinimizeGadget | _
                #PB_Window_MaximizeGadget | _
                #PB_Window_ScreenCentered)

Repeat
        Event = WaitWindowEvent()

        If Event = #PB_Event_CloseWindow
                End
        EndIf
ForEver
```