Hollywood Programming

Class 2: Intermediate level
Object-Oriented Hollywood
github.com/SamuraiCrow/AmiWest2022

Presented by Samuel D. Crow

Tables

- Implemented using Hashing
- Used as Arrays
- Used as Classes
- Used as Namespaces
- Used as File Buffers

Hash Table Traits

- Keys can be integers or strings
- One unique key for every value
- Values can be any type including other tables!
- Commonly used in other languages

Used as Arrays

```
Local x={“abracadabera”, 5, 6.3, {1,2,3}}
Print(x[0])
abracadabera
Print(x[3][2])
3
```

Jagged Arrays

```
Local y={{1},{2,3},{4,5,6}}
```

- Note that the inner arrays are of a different length from each other
- Note also that they are nested tables and not multidimensional
- Real “memory slab” arrays may be coming in a future version of Hollywood

Classes

- Contain methods as well as data
- The difference between a method and function in Hollywood is a reference to “self” is passed in implicitly during a method call
- Can also contain static functions but use a slightly different syntax from methods

Classes Example

```
Local me={}
// This is a method
Function me:new()
   self[“x”]=0
EndFunction
// This is a function
Function me.show()
   Print(self.x)
EndFunction
```

Classes Error

```
Local me={}
// This is a method
Function me:new()
   self[“x”]=0
EndFunction
// This is a function
Function me.show()
   Print(self.x) // <- no self in function
EndFunction
```

Classes Fix

```
Local me={}
// This is a method
Function me:new()
   self[“x”]=0
EndFunction
// This is another method
Function me:show()
   Print(self.x)
EndFunction
```

Initialized Data

```
Local me={
   x=0
   y=x
}
Function me:show()
   Print(self.y)
EndFunction
```

Copy Table Bug

```
Local me={
   x={0}
   y=x
}
Function me:show()
   self.y[0]=1
   Println(self.x[0])
EndFunction
1
```

CopyTable Function

```
Local me={
   x={0}
   y=CopyTable(x) // Deep copy by default
}
Function me:show()
   self.y[0]=1
   Println(self.x[0])
EndFunction
0
```

Polymorphism Part 1

```
Global b={}
b.def={
   x=0
}
Function b.def:show()
   Print(x)
EndFunction
Function b.new()
   Return CopyTable(b.def)
EndFunction
```

Polymorphism Part 2

```
Global c={}
c.def={}
Function c.def:show()
   Print(“nothing to show”)
EndFunction
Function c.new()
   Return CopyTable(c.def)
EndFunction
```

Polymorphism Part 3

```
Global d={b,c}
ForEachI(d, Function(key, value)
   Local x = value.new()
   x:Show()
EndFunction)


0
nothing to show
```

What the ForEachI Function Does

```
For key, value In IPairs(d)
   Local x=value.new()
   x:show()
Next
x=Nil
```

ForEachI Function Part 2

```
For key, value In IPairs(d) Do Block
   Local y=Function(v)
      Local x=v.new()
      x:show()
   EndFunction
   y(value)
EndBlock
```

File Buffers

- Serialize using WriteTable()
- Deserialize using ReadTable()
- Serialization Gotchas
  - Functions serialized as bytecode
  - Functions in tables usually have lowercase keys
  - JSON versus Binary format

File Buffers - Functions

```
Function KillFuncs(table)
   For key,value In Pairs(CopyTable(table, False))
      Switch GetType(value)
         Case #FUNCTION:
            table[key]=Nil
         Case #TABLE:
            table[key]=KillFuncs(value)
      EndSwitch
   Next
EndFunction
```

Why Not Leave Bytecodes In Data?

- Reasons you don’t leave functions in tables
  - Method bugs become embedded in data files
  - Debugged methods are overwritten by buggy versions from ages past
  - Possible attack vector in terms of security
  - Data files become bloated
- Laziness is the only reason to leave them in

Reinserting Methods When Loading

- Mark each class type before killing functions
  - Can be as simple as making a key named “class” and assigning it a string key
  - Write a header and version number in front of the table instead
- Key names of functions and methods stored in files are sometimes unexpectedly case-sensitive

Method Naming Examples

- Method definitions:

```
Function d:FuncName() ;always lowercase
Function d.FuncName() ;also lowercase
```

- When calling the method:

```
d:FuncName() ;FuncName becomes lowercase
```

- Mixed case calls not supported in Hollywood

Unrecognized by Function Call

```
d.FuncName = Function() ;becomes lowercase name
d:FuncName = Function() ;Illegal syntax!
d.FuncName = Function(self) ;lowercase method
d[“FunctionName”] = Function(self) ;mixed case
```

- This syntax bypasses metamethods (notice the key name is already lowercase):

```
RawSet(“d”, “functionname”, Function(self) …
```

Binary vs. JSON

- Binary built-in is generally smaller and faster
- Binary can only be viewed from a hex editor or a specialized viewer made in Hollywood
- JSON (called default) can be edited in a text editor
- JSON is designed for JavaScript compatibility
- Deserializers are available as plug-ins (XML)

Seventh Inning Stretch

You've completed the object-oriented section of the Hollywood programming session.

The next section refers to the source code at:
github.com/SamuraiCrow/RapaEdit