Part 18 - Duck typing

Part 18 - Duck Typing

Definition: Duck Typing

Duck typing is a humorous way of describing the type non-checking system. Initially coined by Dave Thomas in the Ruby community, its premise is that (referring to a value) "if it walks like a duck, and talks like a duck, then it is a duck".

Even though Boo is a statically typed language, Duck Typing is a way to fake being a dynamic language. Duck typing allows variables to be recognized at runtime, instead of compile time. Though this can add a sense of simplicity, it does remove a large security barrier.

Duck Typing Example

```
d as duck
d = 5 // currently set to an integer.
print d
d += 10 // It can do everything an integer
does.
print d
d = "Hi there" // sets it to a string.
print d
d = d.ToUpper() // It can do everything a
string does.
print d
```

Output

```
5
15
Hi there
HI THERE
```

Duck typing is very handy if you are loading from a factory or an unpredictable dynamic library.

> ✅ **Recommendation**
>
> Do not enable duck typing by default. It should only be used in a few situations.

On a side note, The `booish` interpreter has duck typing enabled by default. This can be disabled by typing in `interpreter.Ducky = false`

Here is a practical example of where duck typing is useful.

Practical Duck Typing

```
import System.Threading

def CreateInstance(progid):
    type =
System.Type.GetTypeFromProgID(progid)
    return type()

ie as duck =
CreateInstance("InternetExplorer.Application
")
ie.Visible = true
ie.Navigate2("http://www.go-mono.com/monolog
ue/")

while ie.Busy:
    Thread.Sleep(50ms)

document = ie.Document
print("$(document.title) is
$(document.fileSize) bytes long.")
```

Exercises

1. Come up with another good example where duck typing is effective.

Go on to **Part 19 - Using the Boo Compiler**