Part 15 - Functions as Objects and Multithreading

Part 15 - Functions as Objects and Multithreading

Having Functions act as `objects` exposes three very useful methods:

1. `function.Invoke(<arguments>) as <return type>`
2. `function.BeginInvoke(<arguments>) as IAsyncResult`
3. `function.EndInvoke(IAsyncResult) as <return type>`

`.Invoke` just calls the function normally and acts like it was called with just regular parentheses ().
`.BeginInvoke` starts a seperate thread that does nothing but run the function invoked.
`.EndInvoke` finishes up the previously invoked function and returns the proper return type.

example of .Invoke

```
def Nothing(x):
    return x

i = 5
assert 5 == Nothing(i)
assert i == Nothing.Invoke(i)
assert i == Nothing.Invoke.Invoke(i)
```

Since `.Invoke` is a function itself, it has its own `.Invoke`.

Here's a good example of `.BeginInvoke`

Multithreading with .BeginInvoke

```
import System
import System.Threading

class FibonacciCalculator:
    def constructor():
        _alpha, _beta = 0, 1
        _stopped = true

    def Calculate():
        _stopped = false
        while not _stopped:
            Thread.Sleep(200)
            _alpha, _beta = _beta, _alpha + _beta
            print _beta

    def Start():
        _result = Calculate.BeginInvoke()

    def Stop():
        _stopped = true
        Calculate.EndInvoke(_result)

    _result as IAsyncResult

    _alpha as ulong
    _beta as ulong

    _stopped as bool

fib = FibonacciCalculator()
fib.Start()
prompt("Press enter to stop...\n")
fib.Stop()
```

The output produces the Fibonacci sequence roughly every 200 milliseconds (because that's what the delay is). This will produce an overflow after it gets up to 2^64.
The important thing is that it stops cleanly if you press Enter.

Exercises

1. Think of an exercise

Go on to **Part 16 - Generators**