The Object Oriented Paradigm: What is It?

by Kirby Urner
4D Solutions

for

Thunderbird Early College Charter
(TECC)

March 11, 2008

The art of computer programming has gone from almost impossible, to almost easy, in just three or four generations. During that time, the idea of a “computer language” has emerged, thanks largely to Rear Admiral Grace Hopper (USN), who had a strong grasp of the subject, gave us the idea of a “compiler” (whereas Ada before her gave us the idea of “machine language” (ADA, named for Ada, being the main DoD language by some accounts)).

A big shift in the art of computer programming occurred with the so-called Object Oriented Paradigm or OOP.

Objects shouldn't be given some elusively abstract meaning, at least at first. They're what we consider “things” or “the nouns” of a world, things like waterfalls, traffic signals, planets and monkies.

Early in a TECC student's career, we'd like to introduce Dog and Monkey objects for example, as both subtypes of Mammal. Or consider the class definition below, of a simple Biotum.

File Edit Format Run Options Windows Help

```
"""module:  simplelife.py"""

class Biotum:

    def __init__(self, name):
        self.name = name
        self.stomach = []

    def __call__(self,food):
        self.stomach.append(food)

    def __repr__(self):
        return 'Biotum named ' + self.name
```

Ln: 1 Col: 0

Figure 1: A Biotum Class

In Python, the special name __init__ is triggered on the occasion of birth, in turn signified by the name of a class trailed by parentheses, perhaps containing arguments (see Figure Two below). The __repr__ method fires when any object is required to represent itself, and the __call__ method fires when parentheses directly trail an instance, again perhaps with arguments to pass into it.

We've seen some negative press against “objectifying” (a verb) as a bad thing to do, if that means one lowers one's awareness by

withdrawing any empathy from a situation to merely consider it in some cold light, as means to an end, as “an object of manipulation,” holding oneself aloof. We call such a consciousness “calculating” or “mendacious” or “clinical” or whatever, and sometimes accuse scientists of being this way (sometimes rightly).

That being said, there's plenty of room for anthropic romanticism and/or empathic reasoning, emotional intelligence, within the computer sciences. The dam engineer needs to intuit as well as compute, to feel that water pressure and throughput. Those generators actually turn rather lazily, speedwise, so as not to destroy fish. It's their momentum that's huge. Those magnets would rather not allow steady spinning, resist mightily, but are no match for the Columbia River, and so Portland gets power.

Since Aristotle and before, science has undertaken the job of taxonomizing, of sorting, both flora and fauna. That first cut, into animal, vegetable and mineral, is rather basic to our thinking, with “machine” coming later, along with a deeper understanding of DNA, cell growth, the role of proteins in the construction of body parts, including strong bones and muscles, through a process of metabolism and enzyme-guided synthesis.

Mathematicians have likewise needed to bring order to their worlds, consisting of math objects or math toyz (putting a z on the end signifies their suitability for mature audiences, while reminding us of the importance of play, as much for adults as for children). Examples of math toyz (math objects) would be: vectors, polynomials, integers, sets, functions, complex numbers.

Also basic to rationality, since Plato and before, has been this distinction between the generic thing, and instances of that thing. For example, we speak of a generic “chair” vis-a-vis this or that chair, with all of its special case details. This same distinction carries over into OOP, the Object Oriented Paradigm, wherein the generic thing becomes the “class” and the specific instances its “objects”.

The Chair class would be like a blueprint or design, less any actual chair, although one might delegate a chair to represent the class, perhaps one built by the Shakers or Amish (very parsimonious). Then a given actual chair of actual flesh and blood (ew!), would be an Object or Instance of said Chair class. That's just how the jargon, the shoptalk, usually goes, and it's meant to be intuitive, to mirror how we think outside of any computer language.

Before OOP, one tended to think of instructions and data as the two main components of any programming task. The instructions would be

compiled or “number crunched” into even more compact machine language instructions, while the data would sit “on the heap” ready to be “processed” or “mined” (bussing data to the heap in the first place involved further modeling of memory, into permanent versus faster core storage, but that takes us too deeply into computer anatomy — some other time).

After OOP, the metaphorical vista more closely mirrored the knowledge domain in supplying a familiar population of objects, each with its characteristic state (meaning data) and methods (meaning behaviors). Given this way of thinking stayed anchored in the problem the coder was thinking about anyway, computer or no computer, the code tended to come across as more readable, both to oneself and to other coders. OOP was here to stay, even amidst further emergence and new ferment among paradigms.

A goal of the TECC curriculum is to converge OOP style thinking with mathematical training, such that “dot notation” takes its place amongst others, as one more tool for expressing key algorithms, important recipes in our culture (e.g. Euclid's Algorithm).

Most (not all) object oriented languages use the dot or period to signify a containership relation. The expression mydog.bark() indicates that a noun or thing (mydog) should have its internalized bark method triggered. We know “bark” is a method because of the trailing parentheses, reminiscient of a mouth, and a placeholder for “arguments” or “inputs” (food).

```
File  Edit  Shell  Debug  Options  Windows                Help
>>> import simplelife
>>> imp.reload(simplelife)
<module 'simplelife' from '/home/ki
rby/simplelife.py'>
>>> from simplelife import Biotum
>>> cell1 = Biotum('xV')
>>> cell2 = Biotum('xY')
>>> cell1.stomach
[]
>>> cell1('**')
>>> cell1.stomach
['**']
>>> cell1('&&&')
>>> cell1.stomach
['**', '&&&']
>>> cell1(cell2)
>>> cell1.stomach
['**', '&&&', Biotum named xY]
>>>
>>> |
                                          Ln: 55 Col: 4
```

Figure Two: creating two Biota and feeding them