The Myths of Object-Orientation

James Noble

School of Engineering and Computer Science,
Victoria University of Wellington,
New Zealand
kjx@ecs.vuw.ac.nz

Abstract. Object-Orientation is now over forty years old. In that time, Object-Oriented programming has moved from a Scandinavian cult to a world-wide standard. In this talk I'll revisit the essential principles — myths — of object-orientation, and discuss their role in the evolution of languages from SIMULA to Smalltalk to C++ to Java and beyond. Only by keeping the object-oriented faith can we ensure full-spectrum object-oriented dominance for the next forty years in the project for a new object-oriented century!

In the beginning
So our myths and stories tell us
The programmer created the program
From the eternal nothingness of the void

This talk is about the myths of object-orientation: the myths, the lies that tell the truth — the stories that were here before us, and that we will leave behind. These are the stories we think about when we settle down in our cube to shuffle cards, draw designs on whiteboards, to write tests, to write code, to debug. These are the stories we think about to work out if some program is worthwhile or not, if we will let ourselves be pleased by its shape, to measure our work against the great programmers and designers of the past, whose names reverberate into history and legend. These are the stories we tell our students — at least the ones who have learned to program a little — myths to shape the programs they write; myths to shape the way they think about the programs they write.

1 Abstraction

The founding myth of object-orientation is trinitarian. We hold these truths to be self-evident: an object has state, behaviour, and identity:

- **identity** — an object can be distinguished from all other objects
- **behaviour** — objects communicate by sending messages; these messages are interpreted by the receiving object (i.e. dynamic dispatch)
- **state** — an object has mutable variables, encapsulated fields that can be changed from within the object.

S. Drossopoulou (Ed.): ECOOP 2009, LNCS 5653, pp. 619–629, 2009.

These truths are not, of course, self-evident — they are myths we choose to believe, to act upon, to teach. As a community, we need to share a common language. I claim — offering no support — that these three principles are what we mean when we say "my cat is object-oriented" [1]. Identity allows us to distinguish one object from another, while state and behaviour allow us to build *abstractions* using objects as components. The details of an object's state and behaviour are (or should be) hidden inside that object: an object should own its own implementation. An object's internal state and behaviour should only be visible externally by examining the results returned by message sends.

To the longstanding devotee of objects, ECOOP, and OOPSLA, something big may be missing here — *Hierarchical Program Structures* [2]: that is, *classes* and *inheritance*. While inheritance has been a feature of almost every language since SIMULA's prefixing, some important languages based on prototypes [3,4] lack classes and/or inheritance; the original version of JavaScript is probably the most popular now. Inheritance is not a large part of the story I'm telling today.

To the more recent devotee, something else is missing: *types*. Smalltalk should be a sufficient example that types are not essential to object-orientation!

2 Signification

The Scandinavians have another myth about object-orientation: "*All Programming is Simulation*". In writing up this talk I tried to find the reference, but Google couldn't track it down! When I'm teaching object-orientation these days, I tend to use a slightly longer quote that has the virtue of actually existing:

> *A program execution is regarded as a physical model, simulating the behaviour of either a real or an imaginary part of the world.*
>
> Object-Oriented Programming in BETA [5].

We can unpack this a little:

- **program execution** — the objects and bindings created by running program: the stack and the heap
- **physical model** — the electronic and quantum effects in the CPU and memory hardware that embody the program execution.
- **simulating the behaviour** — a program execution is designed to model, that is, to signify, a referent *outside the program itself*. As the program execution evolves over time, so should the referent.
- **part of the world** — the referent, the part of the world, the business, the context, that the program simulates.
- **real or imaginary** — the referent may predate the program (as in a manual system that is being automated) or it may be created by the program (so has to be imagined by the program's developers).

An object-oriented program is always taken as relating to something — a referent – outside the program itself. The program execution is not the main point of a

program: the program execution must be taken as always referring to some other thing in the world. Here is the program, here is the bank account to which that program refers. The program plays the same part as the old ledger-books the banker's father used to write in with a fountain-pen: the numbers and signs in the ledger *signify* just how much money you have in your account. And as computers infiltrate further and further into society, the program's simulation of the world increasingly takes the place of the referent. If the bank's computer says you have no money, then you have no money, no matter what your documentation or your records or the bank's ledgers (superceded as physical models by the computer) or even your lawyer may say.

One thing — the program — stands for something else — the bank account. A program is a sign, a *semeîon* (since we are in Cyprus), a metaphor, a myth, perhaps a lie — but again, the lie that tells the "truth", that becomes the truth, that embodies the truth.

3 Dirt Is Good

What about the physical model? Is it accidental that OO is a physical model? More recently, I've been rethinking this: is there something in the physicality of it all that is essential?

Ka Mate! Ka Mate! Ka Ora! Ka Ora! We believe in life, in death, in time, in constructors and destructors and garbage collectors, change and decay in all around we see, in mutable state — because these things, this entropy and interconnectedness, is essential in the physical world. And then we leverage this physical world to make models of *itself*.

So what is it about *object-oriented* programming languages that make them good for building models? What is the big divide between object-oriented languages and their contemporary structured counterparts, such as Pascal?

Let's consider object-orientation as "where Pascal went wrong." The big gaps in Pascal (and other structured languages including C) are dynamic memory allocation with `new`, and variant records (the famous hole in Pascal's type safety). The behaviour of `new` and explicit pointers give a semantic model of individually addressable dynamically allocated memory regions; updatable memory gives us object state; and case statements or function pointers give object or type dependent behaviour.

Marxists would say this was a small example of *technological determinism*. Our myths, our aesthetics, our cultures of programming are built bottom up from enabling features of programming languages. Perhaps these features are incidental, or accidental, but it is our myths that make them essential!

4 A Digression

As an aside, other paradigms are not like this: they have other stories, other myths. If object-orientation has "world envy" — we wish to model the world — then other approaches have maths envy, or theory envy, or logic envy: they

want programs to be weightless, to be insubstantial, abstract, zipless, to deny the reality with which they must somehow engage.

Consider the "utter pointlessness" of monads in lazy functional languages. These languages are designed to deny reality, to disavow entropy, to banish time, to reject any causal relationship with the world outside the program, to compute mathematically, with perfect strongly-typed abstractions. They're great for computing factorial functions — but can barely echo back user input.

And yet, the dirt of the world seeps back in — as Freud would say, the repressed unconscious will always return. If we cannot stop the world, we can at least stop the program, millions of times every second if necessary, to interact with the world outside, to read or write that dirty mutable physical state. Or we can contort our programs to simulate the mutability of the physical world.

So, if objects did not exist, we would need to (re)invent them. Consider the history of other programming languages in the last 20 years: Tcl endlessly replicating native C procedures and static data to represent widgets; Newsqueak and parallel Prolog programs using infinite loops to represent objects; and most recently, Erlang playing the same old tricks. As Suad Alagic memorably interjected during Joe Armstrong's ECOOP 2007 keynote: *"that's not a function — that's an object!"*

5 Unification

Physics lives for unification: the grand unified theory of everything is physics' holy grail. We know the unification of magnetism and electricity, of electromagnetism and the weak nuclear force, and the strange duality between waves and particles. In computer science we have our own unification and duality: code and data.

Famously, in Lisp, code and data are the same: programs are lists; data are lists; everything is a list — where list, of course, really means a cons-cell. Is this really an accidental feature of low-level models of computation (Lisp, Turing Machines, Lambda Calculus) and "homoiconic" programming languages [6,7], or something more essential?

Object-orientation goes further than Lisp; we have real data structures; everything is an object (an abstraction) not just a cons-cell; and as for code or data: who can tell? who cares? Abstraction results in unification — object-orientation unifies linked lists and arrays into collections. This is a big theoretical result: In other disciplines they give people Nobel prizes for this!

Self, Eiffel, and C♯, for example, also unify methods with fields and assignments. Java lost this, but reinvented it with JavaBeans and Eclipse's autogenerated accessors, which evolved back into the language as properties in C♯. This abstraction — or unification — is more than syntax: I think object-orientation captures something fundamental about computation, as objects abstract away the differences between data particles that exist in memory space and code waves that propagate in time.

Dynamic dispatch — rather than inheritance — enables this unification, although inheritance makes the code shorter. This is why single dispatch is

essential to object-orientation, because a single dynamic dispatch is enough to ensure that as computation crosses an abstraction boundary the appropriate behaviour ensues. This is why I prefer object-oriented programming (single dispatch on abstract types) to pattern matching (multiple dispatch on concrete types).

The interesting observation here is that syntax precedes semantics; Dahl and Nygaard built simulation systems before any notion of inheritance, and Kay read the machine code of the B5000 file system before designing Smalltalk. Theorising comes along after the concrete artifacts.

Or as Karl Marx (paraphrased in the comic book "Introducing Postmodernism") puts it: "*what we produce is always miles ahead of what we think*" [8]. Myths are the result of our reflecting on our systems and our designs.

6 A Stack Is Not an Object

If object-oriented programming started with Simula, then "object-orientation" as an idea, a principle, a myth, started with Smalltalk. As Alan Kay (who did, after all, win the Turing award for this — as Nygaard and Dahl did later) puts it in the *Early History of Smalltalk* [9] (my emphasis):

> *a new design paradigm—which* ***I called*** *object-oriented.*

A little further on in that chapter, there is another quote:

> *This [object-orientation] lead to the ubiquitous stack data type example in hundreds of papers. To put it mildly, we were amazed at this.*

I was quite amazed with this quote when I first read it, and for several years later I really didn't understand what Kay meant. By that stage I'd been using and teaching object-oriented programming for several years, so of course I thought I understood it. I was especially proud of my example stack object written in Self: a `top` method, a `pop` method, a `push:` method, and I didn't even inherit from `vector`! But here is Alan Kay saying, pretty much, "*A Stack is Not an Object*". Oops.

Fifteen years later I think I understand better what Kay was writing about. A stack is basically a data structure — an abstract data type. A good object should be more than just a data structure: it should represent something outside the program; it should be at a higher level than just a data structure; and it should unify both data and behaviour. In the terms I've used in this talk:

- **signification** — objects should be physical components of a model of something in a world outside the program.
- **abstraction** — objects should represent "higher level goals" rather than applying "procedures to data structures".
- **unification** — objects encompass both state and behaviour (and abstract both simultaneously). This is Kay's "recursion on the idea of the computer itself".

but I can't claim this insight as either novel or original. Towards the end writing out my talk notes for publication, I came across Ralph Johnson's slides for his Object-Oriented Programming and Design course [10,11]. As Ralph describes it:

> *I explain three views of OO programming. The Scandinavian view is that an OO system is one whose creators realise that programming is modelling. The mystical view is that an OO system is one that is built out of objects that communicate by sending messages to each other, and computation is the messages flying from object to object. The software engineering view is that an OO system is one that supports data abstraction, polymorphism by late-binding of function calls, and inheritance.*

Now, this seems rather better than I could manage. I'd just like to hold all three views, simultaneously — as I suspect Ralph does.

7 History, Tragedy, Farce

According to George Santayana (via Google): "Those who cannot remember the past are condemned to repeat it". To paraphrase to Karl Marx (again, and also via Google): "History repeats itself, first as tragedy, second as farce".

This is as true in software as it is in any other human endeavour. We can see it clearly with respect to object-oriented programming languages: we have SIMULA (history) followed by Smalltalk (tragedy) and finally Java (farce). This makes a great party game — choose any area of software and fill in the blanks yourself! Table 1 gives one possible set of answers: some of these are better than others.

Table 1. History, Tragedy, Farce

	History	**Tragedy**	**Farce**
Object-orientation	Simula	Smalltalk	Java
Nested object languages	Simula	BETA	Scala
Smalltalk languages	Smalltalk	Self	Newspeak
Systems languages	BCPL	C	C++
Wirth languages	Pascal	Modula	Oberon
Lisp languages	LISP	Scheme	CommonLisp
ML languages	ML	O'CAML	F♯
Languages beginning with "C"	C	ANSI C	C++
C++ languages	C++	C+@	C♯
C♯ languages	C♯1.0	C♯2.0	C♯ 3.0
BASIC languages	BASIC	VB	VB.net
Orthogonal languages	Algol-68	PL/I	Scala
Computer Companies	DEC	Sun	Oracle
Haskell	Haskell	Haskell	Haskell

8 The Power of $1\frac{1}{2}$

This leads me to the original design principle behind Java. I call it the principle of $1\frac{1}{2}$ — although perhaps the $1+\mathcal{N}$ or $1+\infty$ principle would be fairer, it would be less humourous and thus less memorable!

Java is not a symmetrical language. Coplien has argued symmetry-breaking is a feature of C++ [12]: I'm not so sure. In three important places, Java's design has one first class component, and then a list of second class components (that's the half). So borrowing directly from C++, Java has dynamic single dispatch (message sending) on one function argument to the left of the dot, combined with static multiple dispatch (overriding) on any number of arguments to the right of the dot.

Java's inheritance design follows this scheme: a class `extends` one first class parent — its superclass — and then `implements` any number of other second class interfaces.

Java 1.5 generics also follow this pattern. A generic type has one first class component — the underlying raw type — and then any number of type parameters that are erased at runtime. Overall, an accidental corner case of C++'s design governs much more of the design of Java.

9 Terroir

I recall, as a graduate student, having several "discussions" with Brian Boutel (then head of department) who strongly objected when — in a weak moment, I claimed *"I believe in object-orientation!"* Yea, verily, I had taken Alan Kay into my heart. While Brian was willing to concede a belief in objects may be useful in the practical art of getting real software built, as a researcher he thought that myths should be treated with a certain scepticism — as working hypotheses, not personal beliefs. On reflection, I've come to see that he was right: myths are lies we choose to believe in, knowingly and willingly. As academics, we interrogate them; as researchers, we manufacture them. On further reflection, I remember that Brian was a member of the first Haskell committee. So perhaps he had his own myths too, and was more evangelical about them than he let on.

When I was learning Smalltalk, Brian also used to complain about "Californian" programming — no types, dynamic dispatch, a relaxed interactive programming environment — much warmer, and much less bracing, than Oregon or Glasgow that gave birth to his beloved Haskell. So I wonder if, like wine, do programming languages have terroir? What influence does the environment that nurtures a programming language, or a programming principle, or a myth, have on the result? Smalltalk is Californian, Dick Gabriel has described how Unix (and C) comes from the Bell Labs engineering culture [13], but what of the rest? Can we see the clarity and austerity of Scandinavian fiords in the design of SIMULA? The interlocking relations of a social-democracy in the design of BETA? The Swiss sense of precision and cleanliness in the designs of Pascal (the Mondaine railway clock); Modula (a Tag Heuer watch); and Oberon (Swatch).

How much does a country, a city, a computing department in a city, or the bar where the graduate students drink affect the languages we all end up using?

When I was in Aarhus one year, the graduate students proudly showed me the bar (so they said) preferred by Kristen Nygaard, Bjarne Stroustrup, Anders Hejlsberg, and Mads Tofte — and where most of the best ideas in programming languages had been invented.

Bars after conferences can be good as well: Extreme programming, Aspect-Oriented programming, and Design Patterns (at least) all came from drinking sessions after conferences. And isn't that why we're all here: — to listen to the talks, to Google stuff, and to wake up with next year's ECOOP submission? I still have a vivid memory of drafting the Flexible Alias Protection paper, on a plane home, after drinking with Jan at OOPSLA '97.

Which leaves us only with the question of the origin of Java. This story — involving yet another trinity — has been relegated to the appendix, and for this, I offer neither apology or explanation.

Acknowledgements. Thanks to Jan Vitek for inviting me to give the banquet talk at ECOOP 2008, and for writing the introduction to this version; to the Bouzouki players and glass-balancers for lending me the stage for five minutes; to the English tourists for not throwing anything; to Peter Dickman for the vodka afterwards; to Ewan Tempero for comments on drafts; and to Sophia Drossopoulou for her encouragement and for printing this in the next year's proceedings.

References

1. King, R.: My cat is object-oriented. In: Object-Oriented Concepts, Databases, and Applications, Addison-Wesley, Reading (1989)
2. Dahl, O.J., Hoare, C.A.R.: Hierarchical program structures. In: Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.) Structured Programming, Academic Press, London (1972)
3. Taivalsaari, A.: A Critical View of Inheritance and Reusability in Object-oriented Programming. PhD thesis, University of Jyväskylä (1993)
4. Taivalsaari, A.: Classes vs. prototypes: Some philosophical and historical observations. In: Noble, J., Taivalsaari, A., Moore, I. (eds.) Prototype-Based Programming: Concepts, Languages and Applications. Springer, Heidelberg (1999)
5. Lehrmann Madsen, O., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, Reading (1993)
6. Kay, A.C.: The Reactive Engine. PhD thesis, University of Utah (1969)
7. Mooers, C., Deutsch, L.: Programming languages for non-numeric processing—1: TRAC, a text handling language. In: Proceedings of the 1965 20th ACM National Conference, pp. 229–246. ACM Press, New York (1965)
8. Appignanesi, R.: Introducing Postmodernism. Icon Books Ltd (1999)
9. Kay, A.C.: The early history of Smalltalk. In: Wexelblat, R.L. (ed.) History of Programming Languages Conference (HOPL-II). ACM Press, New York (1993)
10. Johnson, R.: Object-oriented programming and design (2008), `http://st-www.cs.uiuc.edu/users/johnson/598rej/`

11. Johnson, R.: Erlang, the next Java (August 2007), http://www.cincomsmalltalk.com/userblogs/ralph
12. Zhao, L., Coplien, J.: Understanding symmetry in object-oriented languages. Journal of Object Technology 2(5), 123–134 (2003)
13. Gabriel, R.P.: LISP: Good news, bad news, how to win big. AI Expert 6(6), 30–39 (1991)
14. Kipling, R.: Just So Stories. Macmillan, Basingstoke (1902)

The Sing-Song of Old Man Java

With apologies to Rudyard Kipling [14].

Not always was Java as now we do behold him, but a Different Language with very short types. He was small and he ran slowly, and his hype was inordinate: he danced on a TV set in the middle of California, and he went to Big God Gosling.

He went to Gosling at six before breakfast, saying, 'Make me different from all other languages by five this afternoon.'

Up jumped Gosling from his fortress on the multicore and shouted, 'Go away!'

He was small and he ran slowly, and his hype was inordinate: he danced on a set-top-box in the middle of California, and he went to Middle God Steele.

He went to Steele at eight after breakfast, saying, 'Make me different from all other languages; make me, also, wonderfully popular by five this afternoon.'

Up jumped Steele from his virtual reality and shouted, 'Go away!'

He was small and he ran slowly, and his hype was inordinate: he danced on desktop in the middle of California, and he went to the Little God Gilad.

He went to Gilad at ten before dinner-time, saying, 'Make me different from all other languages; make me popular and able to run **anywhere** by five this afternoon.'

Up jumped Gilad from his office in Palo Alto and shouted, 'Yes, I will!'

Gilad called C♯ — dotNet C♯ — always hungry, just in from Redmond and showed him Java.

Gilad said, 'C♯! Wake up, C♯! Do you see that gentleman dancing in a browser? He wants to be popular and able to run anywhere. C♯, make him SO!'

Up jumped C♯ — Microsoft C♯ — and said, 'What, that hack-rabbit?'

Off ran C♯ — CLR C♯ — always hungry, grinning like a coal-scuttle, — ran after Java 1.0.

Off went the proud Java with his short little types like a bunny.

This, O Beloved of mine, ends the first part of the tale!

* * *

He ran on the desktop; he ran on the mainframe; he ran on the handhelds; he ran on the smartcards; he ran till his bytecodes ached.

He had to!

Still ran C♯ — Standard C♯ — always hungry, grinning like a rat-trap, never getting nearer, never getting farther, — ran after Java 1.2.

He had to!

Still ran Java — Old Man Java. He ran through the widgets; he ran through the phidgets; he ran through the applets; he ran through the servlets; he ran through the Topics of EBJ and MIDP; he ran till his parser ached.

He had to!

Still ran C♯ — ECMA C♯ — hungrier and hungrier, grinning like a horse-collar, never getting nearer, never getting farther; and they came to the J.C.P.

Now, there wasn't any proof, and there weren't any monads, and Java didn't know how to parameterise; so he stood on his types and hacked.

He had to!

He hacked through the Enums; he hacked through the Bignums; he hacked in the deserts in the middle of California. He hacked like Java 1.5

First he hacked typevars; then he hacked raw types; then he hacked wildcards; his types growing stronger; his types growing longer. He hadn't any time for rest or refreshment, and he wanted them very much.

Still ran C♯ — ISO C♯ — very much bewildered, very much hungry, and wondering what in the world or out of it made Old Man Java hack?

For he hacked like Haskell; or Eiffel or Ada; or GJ or Scala or Meta-O-CAML.

He had to!

He hacked up variance; he hacked his invariants; he stuck out modules for a balance-weight behind him; and he hacked up type inference too.

He had to!

Still ran C♯ — Dot Net's C♯ — hungrier and hungrier, very much bewildered, and wondering when in the world or out of it would Old Man Java stop.

Then came Gilad from his conference in Cyprus, and said, 'It's five o'clock.'

Down sat C♯ — ECMA Standard C♯ — always hungry, dusky in the sunshine; hung out his tongue and howled.

Down sat Java — Old Man Java — stuck out his types like a milking-stool behind him, and said, 'Thank goodness that's finished!'

Then said Gilad, who is always a gentleman, 'Why aren't you grateful to Microsoft's C♯? Why don't you thank him for all he has done for you?'

Then said Java — Tired Old Java — 'He's chased me out of the VMs of my childhood; he's chased me out of my pluggable semantics he's altered my classpath so I'll never get it back; and he's played Old Scratch with my types.'

Then said Gilad, 'Perhaps I'm mistaken, but didn't you ask me to make you different from all other languages, as well as to make you able to run anywhere? And now it is five o'clock.'

'Yes,' said Java. 'I wish that I hadn't. I thought you would do it by proofs and incantations, but this is a practical joke.'

'Joke!' said Gilad from the bar in the lobby. 'Say that again and I'll whistle up C♯ and run your primitive types off!'

'No,' said Java. 'I must apologise. Types are types, and you needn't alter 'em so far as I am concerned. I only meant to explain to Your Lordliness that I've had nothing to drink since morning, and I'm very thirsty indeed.'

'Yes,' said C♯ — ISO C♯ — 'I am just in the same situation. I've made him different from all other languages; but what may I have for my tea?'

Then said Gilad from the lobby at the conference, 'Come and ask me about it tomorrow, because I'm going to eat.'

So they were left in the middle of Cyprus, Old Man Java and Microsoft C♯, and each said, 'That's your fault.'