Chris M Hostetter
12547674
CS 263
052398

Survey of Object Oriented Programming Languages

1 Introduction

This paper was intended as a learning experience for the author, in an attempt to better understand the history and development of Object Oriented Programming Languages. The research involved in this paper began with a variety of papers which dealt with the definition of "Object Oriented" and from there spread out into several historic papers on the origins of Object Oriented Programming in Simula and Smalltalk. Further reading involved mainstream reference material on many of the more widely used Object Oriented Languages, in addition to historic documents on their design and development.

Due to time constraints, this paper never achieved the scope that the author had intended for it, and much of the research done is not reflected in these pages.[1] This paper only attempts to look at the meaning of "Object Oriented" as a property of Programming Languages, as well as to examine the history, and features of eight particular Object Oriented Languages. It is divided into four sections:

1. **Introduction:** This section.
2. **Notion of Object Orientedness:** Discussion of various authors feeling on OOP.
3. **Some Languages:** A look at the history, development, design, and features of predominant OOPLs.
4. **Research References:** References used for this research project.

2 Notion of Object Orientedness

What does it mean for something to be object oriented? What exactly is an "Object Oriented Programming Language?" These are questions that no one can answer very well. As Tim Rentsch said:

> "*...object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.*" [Rentsch82]

[1] But in theory, what really matters is what the author learned.

Generally speaking, Rentsch was right not only about the 1980's, but the 1990's as well. With all the various "Object Oriented Programming Languages" and "Object Oriented Processes" and "Object Oriented Basket Weaving" in the world today, it seems that establishing a clear definition of what it means to be object oriented, is impossible.

It would appear that instead of attempting to find a coherent definition, a better approach is to accept the idea that programming languages, or processes (or widgets) can not "Be Object Oriented." Instead, we should consider object orientedness as a "notion" which can be associated with systems, programs, languages, etc... Or similarly, as a property that something can exhibit. But just because one facet of a system has object oriented properties, does not mean that the system as a whole (or even other potions of the system) is object oriented.

When dealing specifically with programming, and programming languages, it is important to keep in mind 3 principles:

- A design can be Object Oriented, even if the resulting program isn't. [Madsen88]
- A program can be Object Oriented, even if the language it's written in isn't. [Madsen88]
- An Object Oriented program can be written in almost any language, but a language can't be associated with object orientedness unless it *promotes* Object Oriented Programs. [Stroustrup91]

But all of this side steps the issue that people ultimately want to understand: "What makes programming languages object oriented?" To address this question, consider some excerpts (from an assortment of papers) that attempt to clarify the issue:

> "*The first principle of object oriented programming might be called* intelligence encapsulation: *view objects from outside to provide a natural metaphor of intrinsic behavior. ... It follows that there is no way of opening up an object and looking at it's insides, or updating ("smashing") its state. What is more important is that the concept of opening up an object does not exist in the language.*" [Rentsch82]

> "*... a message is a request of what the sender wants with no hint or concern as to what the receiver should do to accommodate the sender's wishes. ... This notion, a sort of* call by desire*, is central to the object oriented philosophy*" [Rentsch82]

> "*... attributes can be shared by a group while allowing for individuals within the group to reinterpret some "shared" behavior* as it applies to the individuals themselves. *... The result of allowing individual variability is that, given something close to what you want, it is easy to produce exactly what you want by overriding shared behavior with individual behavior - to* adapt" [Rentsch82]

> "*... the entire thrust of its design has been to supercede the concept of data and procedures entirely; to replace these with the more generally useful notions of activity, communication, and inheritance.*" Alan Kay in [Rentsch82]

"*If the term "object-oriented programming language" means anything it must mean a programming language that provides a mechanism that supports the object-oriented style of programming well.*" [Stroustrup91]

"*The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time).*" [Stroustrup91]

"Object-oriented programming: *A program execution is regarded as a* physical model*, simulating the behavior of either a real or imaginary part of the world.*" [Madsen88]

"*... Such a language should support:*

1. *Modelling of concepts and phenomena, i.e. the language must include constructs like class,type, procedure.*
2. *Modelling classification hierarchies, i.e. sub-classing (inheritance) and virtuals.*
3. *Modelling active objects, i.e. concurrency or coroutine sequencing, combined with persistency.*" [Madsen88]

"*Object oriented programming: The computing process is viewed (as described above) as the development of a system, consisting of objects (components), through sequences of changing states.*" [Nygaard86]

"*The object-oriented approach combines three properties: encapsulation, inheritance, and organization*" [Nguyen86]

In addition to the above passages, it is important to keep in mind some items which may (misleadingly) seem object oriented:

"*Object oriented programming is* not *programming using a Simula-like class concept, just as structured programming is not GOTO-less programming.*" [Rentsch82]

"[Simula...] *... also included many "features" such as INSPECT and IN, which are contrary to the object oriented philosophy.*" [Rentsch82]

"*Consider a language having an inheritance mechanism without virtual functions or methods. Could that language be said to support object-oriented programming? I think not...*" [Stroustrup91]

"*Thinking object-oriented does not have to exclude functional expressions when that is more natural. Functions, types and values are in fact needed in order to describe measurable properties of objects.*" [Madsen88]

To summarize these ideas, we can say that following features are object oriented properties – and that Programming Languages which actively support them, can be associated with object orientedness.

- Objects...
  - Autonomous entities
  - No direct access to their components, or run time type[2]
  - Interact purely by messages, with no assumption of implementation
- Object Organization...
  - Inheritance mechanism to construct an object classification hierarchy
  - Not just for code sharing
  - Must have abstract nodes in hierarchy for more realistic modeling
  - Provide potential for individuality in objects
  - Dynamicly determine (via run time type) correct response to messages
- Programs as Models...
  - Programs model the development of some planned system
  - Changing state of system reflected by the changing state of the objects
  - Objects can be thought of as acting concurrently
  - Support for "non OO" programming techniques in situations where they make sense

3 Some Languages

3.1 Simula

3.1.1 Creators

Developed at The Norwegian Computing Center by Kristen Nygaard and Ole-Johan Dahl.

3.1.2 Influences

Algol 60 (as a subset).

[2]It's interesting to note that although most of the literature agrees that a feature to inspect the type of an object at run time is contrary to the OO philosophy, the same sources all agree that dynamic dispatch is a must - but the two features are equivalent.

3.1.3 Development Time-line

In 1961 Nygaard was working for the Norwegian Computing Center (NCC), doing work with simulations and system analysis. Nygraad felt that instead of using existing tools, the best way to approach simulation programming, would be to have a special purpose simulation programming language, that could be used to model systems easily. With his background in simulations, he teamed up with Dahl (an experienced programmer with experience in language design) in 1962.

Nygaard and Dahl proposed their idea for a simulation language in 1963, and although their ideas did not receive much enthusiasm, political issues at the NCC resulted in a contract between the NCC and UNIVAC to provide a Simula implementation and compiler by 1965 – NCC did so, and the result is known as "Simula-I". Refinements were made to Simula I and in 1967 "Simula-67" was released. The most recent standard is Simula-87.

3.1.4 Features and Design

Simula started out as an activity/process based programming language, in which different types (and behavior) of activities are declared, and then multiple processes can be created to carry out different activities. The key power in this original design, was that in addition to having lists of actions to be performed, processes were also data structures, and activities has associated methods. These activities and processes had so much use besides just that of simulation, that when Simula-67 was released, they had been renamed "classes" and "objects" (thus the birth of "object oriented" programming).

In addition to all the features of Algol 60, Simula added support for Objects (as closures which return references to themselves) with protected state, single inheritance for sub-typing and code sharing, partially abstract classes, method overriding, and nested closures (including nested procedures, classes, and classes local to procedures).

Its interesting to note that as the father of object oriented languages, Simula-67 does not support dynamic dispatch – something most people consider necessary for "true" object oriented programming. Instead, an object must be downcast (error checked at run time) and the appropriate attribute/method can then be accessed.

3.2 Smalltalk

3.2.1 Creators

Developed by Alan Kay, Dan Ingalls, and Adele Goldberg at Xerox PARC.

3.2.2 Influences

B220 Tapes, Sketchpad, Simula, and Lisp.

3.2.3 Development Time-line

In 1961, Alan Kay was a programmer for the Air force, and noticed that someone had designed a system for "Burroughs 220" magnetic tapes in which the data on the tape could be

of arbitrary format and size, the beginning of the tape contained the actual code necessary to extract the particular type of date (This would eventually come to be known as encapsulating code and date into objects).

In 1966, Kay went to graduate school at the University of Utah, on his first day he was handed a packet of orientation information – which included a paper on "Sketchpad" by Ivan Sutherland – and found on his desk a pile of tapes and printouts with the note "This is the Algol for the 1108. It doesn't work, Please make it work."

Starting with the packet of information, Kay was fascinated by Sutherland's paper. Sketchpad was a revolutionary graphics workstation (for it's time) which among other things: enabled the user to create "master drawings" and from a master, create multitudes of "instance drawings" with their own distinct characteristics. In addition, the data structures used by Sketchpad were similar to the B220 tapes he had seen in the Air Force, with embedded pointers for modifying the structure.

Kay eventually got to work on the "Algo" printouts on his desk, and was thoroughly confused, there were many constructs he had never seen before and "the documentation read like Norwegian translated into English."[Kay93] What Kay had inadvertently been given was a copy of Simula-I. Looking over the printouts, and reading the documentation, he realized the immense potential of programming with constructs like Simula's activities and processes, or Sketchpad's masters and instances. Kay quickly developed a vague idea for programming by not only breaking the problem down into smaller sub problems, but by breaking the computer down into thousands of smaller simulated computers (or objects) to solve all of the sub problems.

Kay began working on "personal computers" with an eye towards an "object oriented" interface. In 1970 left Utah to work for Xerox PARC, and began developing a desktop computer or use by children called "KiddiKomp" (later "miniCOM") which had a combination programming language / user interface called Smalltalk(-71) to stress it's ease of use.

Smalltalk was redefined from scratch in 1972 on a bet, (that Kay could define the "most powerful language in the world" in "a page of code") and Smalltalk-72 has since been considered the first "real Smalltalk". Smalltalk was redesigned again in 1976 by Dan Ingalls, and then again in 1980 just before it was released to the public.

3.2.4 Features and Design

Smalltalk is an untyped, class based language. To better model the difference between performing actions on types of objects, and on a distinct object, Smalltalk classifies attributes and methods as belonging to either the class or to instances. Class methods are located in the class meta-object and can only refer to class attributes (also located in the meta-object to provide shared state for all instances). Instance methods are kept locally to each object and can refer to both the class variables and the instance variables (which provide local state) All methods are public, while all attributes are private.

Single inheritance is provided, along with (partially) abstract classes, and method overriding (including signature modification). Although Smalltalk is untyped, the main purpose of inheritance is not just code sharing. The underlying principle is similar to subtyping in that programmers should use inheritance to provide specialization of objects, and

can subsequently feel secure that it will be safe to use an instance of a subclass as if it were an instance of the superclass – without having to worry about run time errors.

3.3 Modula-3

3.3.1 Creators

Designed by Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, Greg Nelson. The specification was written by Lucille Glassman and Greg Nelson .

3.3.2 Influences

Modula-2+ (and from it: Modula-2, Mesa, Cedar).

3.3.3 Development Time-line

In November of 1986 Maurice Wilkes proposed that the ideas in Modula-2+ be formalized into a new standardized language in the Modula Family. As a result the Modula-3 committee was formed by Digital Equipment Corporation in cooperation with the Otivetti Research Center. The initial language definition was published in August of 1988, and then revised (based on the recommendations of implementors) in January of 1989.

3.3.4 Features and Design

Modula-3 is a class based language in which class names act as type names – there are no explicit type definitions or declarations.

Class definitions are "partial opaque" - meaning that methods and attributes may or may not be visible to other classes. Single Inheritance is provided as a means of type specialization, (allowing sub-classes to override the methods of their super-class) which also provides code reuse, and Abstract Classes are allowed as a means to specify types (which contain only method declarations and signatures) without implementations.

In addition to inheritance as a method of class specification, Modula-3 provides Generic Modules which are templates parameterized by types. Generics are not polymorphic, and provide only source code reuse (not target code reuse).

Modula-3 is strongly typed, with no automatic conversion, or type inference. In addition, Modula-3 defines type equality based on type/class structure, and not type/class name:

> "*Two types are the same if their definitions become the same when expanded; that is, when all constant expressions are replaced by their values and all type names are replaced by their definitions. In the case of recursive types, the expansion is the infinite limit of the partial expansions.*" [Nelson91]

Beyond Modula-3's Object Oriented features, it also provides Lightweight Threads, Exceptions, Modules, and Module Interfaces (similar to C header files, but more restrictive). The language definition also provides a means for programmers to override the compilers safety checks on particular modules by declaring them Unsafe.

3.4 Self

3.4.1 Creators

David Ungar and Randall Smith.

3.4.2 Influences

Self was most heavily influenced by Smalltalk, but borrows from a wide variety of prototype based research languages (specifically "O" by Peter Deutsch).

3.4.3 Development Time-line

Self was initially designed in 1986, and the language design was released in 1987. The first public implementation was released by Stanford in 1991, the most recent release is Self 4.0 from Sun Microsystems in 1995.

3.4.4 Features and Design

Self is a Classless Language, which uses prototype objects, and cloning to construct new objects. The run-time environment is responsible for performing dynamic type checking, and there are no Static types, or type declarations required (or even included) in the language.

The fundamental principle of Self is "Messages-at-the-Bottom". All operations are implemented as messages. Every object is composed entirely of slots, which contain either state, or behavior. When a message is received by an object, the slots of the object are checked for the corresponding message. If the message is not found in one of the objects slots, then the pointer in the slot named "parent" (which every object has as a result of cloning) is accessed, and the message lookup on the slots recurses up the parent pointers.

Once the appropriate slot is found, the contents of the slot are either a pointer to another object (a variable) or a method. methods can be thought of as "prototype activation records" when a slot containing a prototype activation record is accessed, it is cloned (into an actual activation record), and the a pointer to the object which was sent the original message is stored in the "parent" slot (which can be thought of as a self pointer for the scope of the method)

In this manner, inheritance, shared state, code sharing, and dynamic dispatch are all provided via slots, messages, and cloning.

In addition, each slot of an object can be thought of as memory address which can be reassigned at will. Thus not only can methods be modified in the middle of program execution, but attributes can be changed into methods, and vice-versa.

3.5 Eiffel

3.5.1 Creators

Eiffel was designed by Bertrand Meyer at Interactive Software Engineering.

3.5.2 Influences

The OO aspects of Eiffel were directly influenced by Simula67, while the "Design By Contract" aspects of the language were heavily influenced by Meyer's earlier academic work in software verification.

3.5.3 Development Time-line

The initial ideas for Eiffel were conceived in September of 1985, and subsequently released to the public (as ISE Eiffel 1) in October of 1986. Eiffel is still evolving, and the latest version is ISE Eiffel 4.

3.5.4 Features and Design

Eiffel is a Class based language, in which the definition of "Type" and "Class" are identical. Type equivalence is based on Class name equivalence. [3]

Classes may contain (multiple) Feature clauses which can in turn contain multiple Attributes/values and Routines/procedures. The classification of a given feature (Routine or Attribute) is unknown to other classes, ie: an Attribute of type T has the same "appearance" as a Routine which takes no arguments and returns an item of type T. Each Feature list has an associated Client list specifying the Classes (of objects) which are permitted to access those particular features. Classes may also Defer the implementation of any feature, making it (and any sub-class which does not provide an implementation for each deferred feature) an Abstract class.

Genericity classes are class templates declared with Formal Generic Parameters which may be constrained by a super-class. Instantiating a Generic class requires an Actual Generic Parameter (which is a sub-class of the specified constraint) for each formal parameter. These Generic classes provide source code reuse, but not target code reuse.

Eiffel supports multiple inheritance (including code reuse) with compiler enforced Renaming of conflicting features. In addition, Eiffel allows the programmer to not only Redefine (or Undefine) the implementation of particular features, but also modify the Client list of inherited features.

An interesting "feature" of the Eiffel specification, is that feature Redefinition is covariant. The reasoning is that for most situations, covariance is more useful then contravariance, and even though the language specification allows the possibility of the run-time type errors, it is the responsibility of the compiler to catch these situations. But, as the Eiffel FAQ admits:

> "*no compiler available today implements full static type checking. Some insert run-time checks.*" [Arnaud98]

One of the most predominant features of Eiffel is built in support for "Design By Contract", which is not necessarily object oriented, but works well with object oriented design processes. Eiffel provides Assertions in the form of Invariants over objects, and

[3] in the case of generic class templates, class name equivalence includes the names of the actual parameters

Pre/Postconditions on individual routines. In general, an Assertion is a boolean expression which has no effect if true, but otherwise results in a Run Time Exception. Preconditions are evaluated prior to the body of the associated routine (in the scope of the routine's body, with the formal parameters bound to the actual arguments). Postconditions are evaluated after the body of the associated routine (again, in the scope of the routine's body, with the formal parameters bound to the actual arguments). Invariants are evaluated after any feature is accessed.

Assertions are extremely powerful when combined with Inheritance. Eiffel requires not only that the Invariants of all super-classes are compatible, but that the Redefinition of any feature must either ad-hear to the initial Pre/Postconditions of the superclass, or have a weaker Precondition and a stronger Postcondition.

3.6 Sather

3.6.1 Creators

Sather was designed at the International Computer Science Institute. The initial designers were Stephen Omohundro, Chu-Cheow Lim, and Heinz Schmidt.

3.6.2 Influences

The major influence to Sather's design has been Eiffel, but it has also been influenced directly by C, C++, Cecil, CLOS, CLU, Common Lisp, Dylan, ML, Modula-3, Oberon, Objective C, Pascal, SAIL, School, Self, and Smalltalk.

3.6.3 Development Time-line

The initial design for Sather ("Version O") was written in the summer of 1990, and released by ICSI to the public in June of 1991 (as version 0.1). While the core language has not changed, new features were added slowly, leading up to the release of Sather 1.0 in the summer of 1994, followed by Sather 1.1 in September, 1995. The main addition of 1.1 was the incorporation of Thread support from pSather (a language which had been evolving in parallel to Sather at ICSI). In addition to pSather, the language Sather-K is a derivative of Sather being developed at the University of Karlsruhe in Germany. Sather-K diverged from Sather when Sather 1.1 was released in 1995.

3.6.4 Features and Design

Sather is a Class based language which borrows a lot of ideas and semantics from Eiffel (such as Generic Classes and Exceptions) but attempts to simplify where ever possible.

Like Eiffel, Sather provides public and private Attributes and Routines (in addition to read-only Attributes). But unlike Eiffel, Sather does not permit explicit access lists for Features.

Sather also provides support for the "Design By Contract" principle in Eiffel, including Pre/Postconditions, Invariants, and generalized Asserts, which are statements that can appear in any block of code and result in a fatal error if they do not evaluate to true.

The biggest difference between Sather and Eiffel is the Inheritance system. In Sather, Inheritance is divided into 2 notions: Sub-typing, and Code Inclusion.

Sub-typing is provided purely as a method of type specialization. A (Concrete) Class my be declared as a Sub-type of an Abstract class, and all objects of the Class then automatically conform to the type of the Abstract class. An Abstract class is only an Interface containing Routine and Attribute type signatures, and does not contain any form of implementation (and hence: can not be instantiated). Each Concrete class may be a Sub-type of at most one Abstract class, but Abstract classes can by Sub-types of any number of other Abstract Classes.

In contrast to Sub-typing, Code Inclusion allows a class to directly import the implementation of other (multiple) classes, for the purpose of code reuse (without any affect on the Includer's type). The Including Class may Redefine, Undefine, Rename, or modify the access permissions of any included Routines or Attribute, with compiler enforced Renaming of name conflicts. Similar to the notion of Abstract Classes with Sub-Typing, there are Partial Classes, which posses no type (and hence: can not be instantiated) but may be included by any number of Concrete Classes

(It may be convenient to think of Abstract Classes as type declarations which multiple Concrete Classes can implement, while Partial Classes are code repositories which can by used by multiple Concrete Classes.)

The most impressive feature of Sather is the inclusion of the type "SAME" which is the "Self Type" discussed in Object Oriented theory, but rarely implemented in actual languages.

3.7 C++

3.7.1 Creators

Bjarne Stroustrup.

3.7.2 Influences

C, Simula, and Cpre (a C preprocessor written by Stroustrup in 1979 to make psuedo Simula Classes).

3.7.3 Development Time-line

"C with Classes" was released in 1980 as an enhanced version of C (implemented using C compilers with a preprocessor) which included Classes for data abstraction. C with Classes was designed so that a preprocessor could make direct conversions from classes to structs, by making member functions global, renaming them to include the class name and modifying the argument list to include a the struct equivalent of the method's class as the first argument.

In 1982 Stroustrup began working an a better version of C with classes which would be "truer" Object Oriented superset of C. In 1983 the first version of C++ was released and more advanced Object Oriented features were rapidly until 1995 when the first commercial version was released. More features (including templates) were continually added until 1989,

at which time C++ obtained some level of stability (An ISO standard version of C++ was finalized in 1998).

3.7.4 Features and Design

C++ is a class based language, designed to allow the programmer very low level control over object structure and access. Object oriented features of C++ include: Virtual (abstract) functions - resulting in virtual classes, public/private/protected access control over individual member functions and attributes, friend classes (for allowing explicitly named classes to access private state), nested classes, multiple inheritance (for sub-typing and code sharing) with method redefinition, and templated (generic) classes and functions.

Other general features include: user controlled memory (the heap), direct memory references, static type checking, method overloading, exceptions, threads, and explicitly constructed namespaces.

Unfortunately the language specification does not go far in explaining the details of how these features interact, and many implementations are incompatible.

3.7.5 Java

3.7.6 Creators

Developed by by James Gosling, Bill Joy, and Guy Steele at Sun Microsystems.

3.7.7 Influences

Modula-3, C++, and Lisp.

3.7.8 Development Time-line

The premise for Java arose from James Gosling in 1991, because of the frustrations he had using C++ to program embedded systems software for "smart" electronics devices at Sun Microsystems. Gosling began developing the language "Oak" to be a safe, object oriented systems language. By 1993 Oak had been renamed "Java" and several prototype electronic devices that had been programmed with Java were available - but the market didn't seem interest. Around this time, the WWW was drasticly increasing in use, and Sun began to see uses for Java's small, safe, platform independent byte code in the online community. During 1994 the language was refined and eventually released in as version 1.0 in 1995. An update (1.1) was released in 1996.

3.7.9 Features and Design

Java is a Class based language, that was originally designed for programming embeded systems. Because of this, the ideas of speed, platform independence, and run time safety are crucial in it's design. As the motivation behind the language shifted to the WWW, the issues of speed, platform independence, and safety remained, but the idea of distributed programs became extremely important. One of the main features Java provides is the Remote Method

Invocation (RMI) system, which allows semi-transparent method invocation and exchange of objects between virtual machines (even across the network)

In addition to the RMI, other general programming features supported are exceptions, garbage collection (which is not only supported - but considered crucial), byte code verification (which validates the safety of a given program), threads, method overloading, and packages (for creating name spaces)

As an Object oriented language, Java supports multiple levels of implementation hiding, partially abstract classes, final classes (which can not be inherited from) and static (class) variables.

Single inheritance of Classes is provided for subtypeing and code sharing, in addition to multiple inheritance of "Interfaces" - which act as Type declarations, or completely abstract classes. As of version 1.1, Java now also supports nested and anonymous classes.

4 Research References

Abadi96 Abadi, Martin and Cardelli, Luca
A Theory of Objects
Springer-Verlag - 1996

Arnaud98 Arnaud, Franck
"comp.lang.eiffel Frequently Asked Questions"
http://www.cis.ohio-state.edu/hypertext/faq/usenet/eiffel-faq/faq.html - 1998

Dahl66 Dahl, Ole-Johan and Nygaard, Kristen
"Simula – an ALGOL-Based Simulation Language"
Communications of the ACM - v9, n9 Ed: D. E. Knuth - September 1966

Dahl72 Dahl, Ole-Johan and Hoare, C.A.R.
"Hierarchical Program Structures"
Structured Programming Ed: Ole-Johan Dahl - 1972

Dugan94 Dugan, Benedict
"Simula and Smalltalk: A Social and Political History"
http://www.cs.washington.edu/homes/brd/history.html - 1994

Freeman95 Freeman, Steve
"Partial Revelation and Modula-3"
Dr. Dobb's Journal - v20, n10 - October 1995

Foote89 Foote, Brian
"Class Warfare: Classes vs Prototypes"
http://laputa.isdn.uiuc.edu/warfare.html - August 1989

Gomes Gomes, B.; Stoutamire, D.; Weisssman, B.; and Klawitter, H.
"Sather 1.1 : Language Essentials"
http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/contents.html

Gosling96 Gosling, James; Joy, Bill; and Steele, Guy
"The Java Language Specification"
http://java.sun.com/docs/books/jls/html/ - 1996

Gosling96-2 Gosling, James and McGilton Henry
The Java Language Environment
Sun Microsystems - 1996

Gosling97 Gosling, James
"The Feel of Java"
Computer Ed: IEEE - 1997

Kay93 Kay, Alan
"The Early History of Smalltalk"
ACM SIGPLAN NOTICES - v28, n3 Ed: Richard Wexelblat - March 1993

Khor95 Khor, Kheng-Khoon; Chavis, Nathaniel; Lovett, Steve; and White, David
"IBM Smalltalk Tutorial"
http://www2.ncsu.edu/eos/info/ece480_info/project/spring96/proj63/www/ - 1995

Knudsen97 Knudsen, Jorgen Lindskov
"The BETA Home Page"
http://www.daimi.aau.dk/~beta/ - 1997

Madsen86 Madsen, Ole Lehrmann
"Block Structure and Object Oriented Languages"
ACM SIGPLAN NOTICES - v21, n10 Ed: G. Richard Wexelblat - October 1986

Madsen88 Madsen, Ole Lehrmann
"What object-oriented programming may be - and what it does not have to be"
Lecture Notes in Computer Science Ed: G. Goos and J. Hartmanis - 1988

Meyer92 Meyer, Bertrand
Eiffel: The Language
Prentice Hall International - 1992

Meyer98 Meyer, Bertrand
"An Invitation to Eiffel"
http://www.eiffel.com/doc/manuals/language/intro/ - 1998

Nelson91 Nelson, Greg
Systems Programming with Modula-3
Prentice Hall Series in Innovative Technology - 1998

Nguyen86 nguyen, Van and Hailpern, Brent
"A Generalized Object Model"
ACM SIGPLAN NOTICES - v21, n10 Ed: G. Richard Wexelblat - October 1986

Nygaard81 Nygaard, Kristen and Dahl, Ole-Johan
"The Development of the Simula Languages"
History of Programming Languages Ed: Richard Wexelblat - 1981

Nygaard86 Nygaard, Kristen
"Basic Concepts of Object Oriented Programming"
ACM SIGPLAN NOTICES - v21, n10 Ed: G. Richard Wexelblat - October 1986

Omohundro96 Omohundro, Stephen
"The Sather language: Efficient, Interactive, Object-Oriented Programming"
http://www.icsi.berkeley.edu/~sather/Publications/article.html - 1996

Rentsch82 Nguyen, Van and Hailpern, Brent
"A Generalized Object Model"
ACM SIGPLAN NOTICES - v17, n9 Ed: G. Richard Wexelblat - September 1982

Stroustrup86 Stroustrup, Bjarne
"An Overview of C++"
ACM SIGPLAN NOTICES v21, n10 Ed: G. Richard Wexelblat - October 1986

Stroustrup91 Stroustrup, Bjarne
"What is "Object-Oriented Programming"? (1991 revised version)"
http://www.research.att.com/~bs/whatis.ps - 1991

Stroustrup94 Stroustrup, Bjarne
The Design and Evolution of C++
Addison-Wesley - 1994

Stroustrup98 Stroustrup, Bjarne
"An Overview of the C++ Programming Language"
http://www.research.att.com/~bs/crc.ps - 1991

Ungar91 Ungar, David and Smith, Randall B.
"Self: The Power of Simplisity"
LISP and Symbolic Computation: An International Journal - v4, n3 - 1991

Winder97 Winder, Russell and Roberts, Graham
"A (Very!) Short History of Java"
http://www.dcs.kcl.ac.uk/DevJavaSoft/Copy/book-7.html - 1997

Wolczko96 Wolczko, Mario and Smith, Randall B.
"Prototype-Based Application Construction Using SELF 4.0"
http://www.cs.ucsb.edu/oocsb/self/release/Self-4.0/Tutorial/ - 1996