# OOP: THE REST OF THE STORY

Chuck Allison
Utah Valley University
chuck.allison@uvu.edu

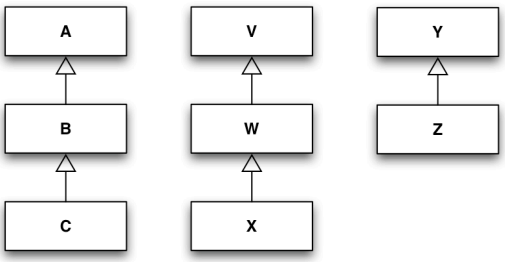
Nathan Liddle
Utah Valley University
nliddle@gmail.com

The object-oriented paradigm has influenced software development for decades now and has found its place in the mainstream of CS curricula. Its facility for improving program organization is well known and today's programmers are expected to facilely employ its features when they apply. Few programmers, and even fewer CS curricula, however, are cognizant of some of OOP's more powerful aspects. Polymorphism, for example, does not only apply to one class hierarchy, hence *multimethods* are part of the OOP repertoire. Software designers should also understand how method pre-and-post conditions interact with inheritance, since class interfaces constitute a *contract* with client programs. Finally, it is important to understand that the features of object-oriented programming do not explicitly require statically defined classes. In this paper we explore multimethods, contract programming, and implementing objects without an explicit class type, with examples in popular, modern programming languages.

## POLYMORPHISM IN MORE THAN ONE DIMENSION

Subtype polymorphism, as taught in typical CS curricula, consists of a *dynamic search* in a *single* class hierarchy for the most specific method consistent with a specific function call. While this is certainly the most common case of subtype polymorphism in object-oriented programs, it is not the only case to consider. The popularity of the Visitor Pattern [1], which implements two-dimensional polymorphism (aka *double dispatch*), illustrates the utility of methods where more than one parameter participates in runtime method selection. Visitor requires explicit cooperation between the two hierarchies, however, and is thus limited to the case of two dimensions.

Common Lisp supports runtime method dispatch with an arbitrary number of dimensions.[2] In Lisp you define a "generic function", a placeholder interface for stand-alone functions that take parameters from multiple hierarchies. Consider the following three sample hierarchies.




Suppose we define a generic function where the first parameter comes from the A-B-C hierarchy, the second from V-W-X, and the third from Y-Z. Given a valid function call, the runtime mechanism must select the *most specific* method where the dynamic type of each

argument has an is-a relationship with its respective parameter in a specific implementation for a combination of parameter types. To determine an ordering for specificity, you can consider the three parameters as the indices of three nested loops traversing the possible type combinations, with the first hierarchy as the set of target objects for the outer loop, the second hierarchy for the next inner loop, and so on. For our sample hierarchy the following sequence of type triples represents the most-general-to-most-specific ordering of possible argument types:

(A,V,Y), (A,V,Z), (A,W,Y), (A,W,Z), (A,X,Y), (A,X,Z),(B,V,Y), (B,V,Z), (B,W,Y), (B,W,Z), (B,X,Y), (B,X,Z), (C,V,Y), (C,V,Z), (C,W,Y), (C,W,Z), (C,X,Y), (C,X,Z)

Now suppose that only the following method combinations actually have implementations:

(A,V,Y), (A,V,Z) , (A,X,Z), (B,W,Y) , (B,X,Z) , (C,V,Z), (C,X,Y)

The correct method can be selected by traversing the second list in *reverse order* until the first combination that has an is-a relationship between each argument and its corresponding parameter is found. For example, the call **f(c,w,z)** will call the (C,V,Z) implementation since a W object "is-a" V. The following complete Common Lisp program shows the runtime bindings for all parameter combinations.

```
;; Define 3 class hierarchies
(defclass A () ())
(defclass B (A) ())
(defclass C (B) ())

(defclass V () ())
(defclass W (V) ())
(defclass X (W) ())

(defclass Y () ())
(defclass Z (Y) ())

;; Define Multimethods
(defgeneric f (p1 p2 p3))
(defmethod f ((p1 A) (p2 V) (p3 Y)) (print '(A V Y)))
(defmethod f ((p1 B) (p2 W) (p3 Y)) (print '(B W Y)))
(defmethod f ((p1 C) (p2 X) (p3 Y)) (print '(C X Y)))
(defmethod f ((p1 A) (p2 V) (p3 Z)) (print '(A V Z)))
(defmethod f ((p1 A) (p2 X) (p3 Z)) (print '(A X Z)))
(defmethod f ((p1 B) (p2 X) (p3 Z)) (print '(B X Z)))
(defmethod f ((p1 C) (p2 V) (p3 Z)) (print '(C V Z)))

;; Create objects
(setf a (make-instance 'A))
(setf b (make-instance 'B))
(setf c (make-instance 'C))
(setf v (make-instance 'V))
(setf w (make-instance 'W))
(setf x (make-instance 'X))
(setf y (make-instance 'Y))
(setf z (make-instance 'Z))

;; Test Output (in 2 columns):
(f a v y)   ; (A V Y)
(f a v z)   ; (A V Z)
(f a w y)   ; (A V Y)
(f a w z)   ; (A V Z)
(f a x y)   ; (A V Y)
(f a x z)   ; (A X Z)
(f b v y)   ; (A V Y)
(f b v z)   ; (A V Z)
(f b w y)   ; (B W Y)
```

```
(f b w z)   ; (B W Y)                    (f c w y)   ; (B W Y)
(f b x y)   ; (B W Y)                    (f c w z)   ; (C V Z)
(f b x z)   ; (B X Z)                    (f c x y)   ; (C X Y)
(f c v y)   ; (A V Y)                    (f c x z)   ; (C X Y)
(f c v z)   ; (C V Z)
```

Not all languages support multimethods, of course. For this particular example, one can hard-code the method selection process by manually searching the implementations in most specific to most general order, as illustrated in the C++ except below:

```
string dispatch(const A& a, const V& v, const Y& y) {
    const B* pb = dynamic_cast<const B*>(&a);
    const C* pc = dynamic_cast<const C*>(&a);
    const W* pw = dynamic_cast<const W*>(&v);
    const X* px = dynamic_cast<const X*>(&v);
    const Z* pz = dynamic_cast<const Z*>(&y);

    if (pc) {
        if (px)
            return "(C X Y)";
        if (pz)
            return "(C V Z)";
    }
    if (pb) {
        if (px && pz)
            return "(B X Z)";
        if (pw)
            return "(B W Y)";
    }
    // The first parameter at this point must be an explicit A
    if (px && pz)
        return "(A X Z)";
    if (pz)
        return "(A V Z)";
    return "(A V Y)";
}
```

Although only applicable to this particular example, this code should help students understand what languages like Lisp have to do behind the scenes to make mutimethods work. C#, Java, Python and Perl have workarounds for programmers who need multi-dimensional polymorphism in those languages.[3]

## ENFORCING POLYMORPHIC METHOD SPECIFICATIONS

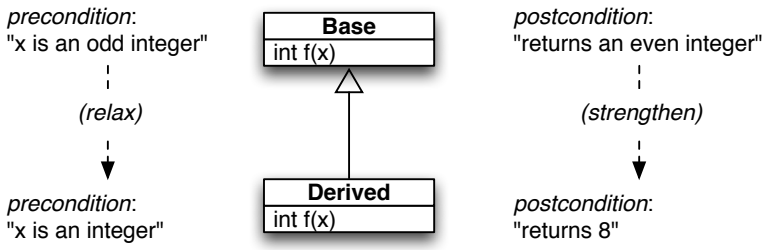
All functions have, in essence, a "contract" with clients. Given certain *preconditions* (e.g., input or expected environmental settings), the function will guarantee specific *postconditions* (e.g., output or side effects).

If the function is an object method, the rules of inheritance interact with the contract. The Liskov Substitution Principle (LSP) specifies that instances of subclasses should be able to substitute for superclass instances without changing the contract. Liskov and Wing noted that this can be achieved through *contravariance* of arguments and *covariance* of return types in the subclasses. [4]

Consider a scheduler class with a method that consults an appointment database to find an available appointment on or after a particular date. The precondition for the base class method could simply be that the date given is a valid future date, while the postcondition

might require that the date returned would be a previously open appointment on or after the date specified. In order for a derived class method to follow the LSP, it would have to accept at least all of the values accepted by the base class method. The output could be more specific, such as finding the next available *weekend* appointment, but it should still provide a date on or after the one specified.

The following diagram depicts a method, **f**, in a class hierarchy where the inputs in the subclass override for **f** are allowed to be more general (contravariance of arguments), and the output is more specific (covariance of return types).




Newer object-oriented languages incorporate some of these rules in the language design. The following D language code shows these conditions in its class declarations:

```
class Base {
public:
    int f(int x)
    in {
        assert(x % 2 == 1);         // Pre-condition
    }
    out (retval) {
        assert(retval % 2 == 0);    // Post-condition
    }
    body {
        int retval = x*2;
        return retval;
    }
}

class Derived : Base {
public:
    int f(int x)
    in {
        assert(is(typeof(x) : int));// Less strict than x % 2 == 1
    }
    out (retval) {
        assert(retval == 8);        // More specific than x % 2 == 0
    }
    body {
        int retval = 8;
        return retval;
    }
}
```

The subclass class above allows a broader range of input than the superclass, and its return type is more specific. The principles of contravariance and covariance for function contracts are summarized in the mnemonic: “Require no more; promise no less.” [5]

## OBJECTS WITHOUT EXPLICIT CLASS TYPES

The essence of object-oriented programming consists of

1. *Encapsulation* (separating client interfaces from implementation detail)
2. *Subtyping* (where one group of objects can be considered a subset of another, and where the functionality of the former is a superset of the latter)
3. *Dynamic Method Binding* (where the most specific, applicable method is chosen for execution)

Often this conceptual triplet is expressed with the terms *classes*, *inheritance*, and *polymorphism*. In most object-oriented languages, however, classes and inheritance are *static* software artifacts; an object's interface and relationship to other types is determined solely by source code processed at compile time. A great deal of flexibility is gained, however, when the behavior of an object is left to runtime determination—*dynamic inheritance*, if you will.

In static object-oriented languages, if class Y inherits from class X, then an instance **y** of type Y can invoke X's methods, and in addition can provide overrides for some or all of the methods inherited from X. Again, the inheritance relationship between X and Y is specified in the *source code* and enforced by the *compiler*. In a *prototypal* language, such as JavaScript [5], an object can be created at runtime to "inherit" properties of another, existing *object*; the latter is called the *prototype* of the former. Whenever an object fails to find a requested method, it delegates the request to its prototype object. There are no classes in prototypal languages; instead, data and methods are bound directly to an object. In the following JavaScript example, the object **x**, which implements a simple stack data structure, acts as prototype for the object **y**.

```
x = {
    data : [],
    push : function (a) {
        this.data.push(a);
    },
    pop : function () {
       return this.data.pop();
    }
};

// Make x y's prototype
var F = function () {}
F.prototype = x;
y = new F();

// Add a top method to y
y.top = function () {
    return this.data[this.data.length-1];
}

y.push(1);
println(y.top())    // 1
y.push(2);
println(y.top())    // 2
```

The relationship between **y** and **x** in this example is established at runtime. In addition, we have dynamically added a **top** method to the object **y**. Methods can also be removed by setting them to **null**.

Besides the awkward syntax of establishing **x** as **y**'s prototype, JavaScript does not directly support changing an object's prototype at runtime. Because Python is universally available, increasingly popular, and eminently readable, we have implemented a simple prototype system in Python using a message-passing protocol, as seen in the following code excerpt. We use a single class (Object) to achieve the desired behavior, but classes are otherwise not used.

```
class Object(object):
    def __init__(self):
        self.prototype = None

    def send(self, msg, *parms):
        if hasattr(self, msg):
            f = getattr(self, msg)
            return f(self, *parms)                    # Plain function call
        else:
            if not self.prototype:
                raise Exception, 'no such method: ' + msg
            return self.prototype.send(msg, *parms)  # Delegate request

    def set_prototype(self, prototype):
        assert(isinstance(prototype,Object))
        self.prototype = prototype

    def clone(self):
        return copy.deepcopy(self)
```

An instance becomes a prototype via the **set_prototype** method, which may be called at any time, allowing an object to change prototypes during the lifetime of a program. Methods are invoked by **Object**'s **send** method, passing the method name as a string along with any required arguments. The recursive nature of **send** supports an arbitrarily long sequence of prototypes. The following code repeats the actions of the JavaScript example seen earlier.

```
# Bind data and methods to x
x = Object()
x.data = []
x.push = lambda this, a: this.data.append(a)
x.pop = lambda this: this.data.pop()

# Make x y's prototype; add method "top"
y = Object()
y.set_prototype(x)
y.top = lambda this: this.prototype.data[-1]

y.send('push',1)
print y.send('top')     // 1
y.send('push',2)
print y.send('top')     // 2
```

## SUMMARY

The object-oriented paradigm offers more power and sophistication than many practitioners realize. Multimethods, contract programming, and dynamic inheritance give software developers useful tools for implementing quality software to solve computational problems. We have found these topics suitable for upper division students, particularly in an analysis of programing languages course.

REFERENCES


1. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
2. Keene, S., *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1989.
3. A nice summary with reference links is found on Wikipedia at http://en.wikipedia.org/wiki/Multiple_dispatch
4. Liskov, B. and Wing, J., "A Behavioral Notion of Subtyping", *ACM Transactions On Programming Languages and Systems*, Vol. 16, No. 6, Nov. 1994, pp. 1811-1841.
5. Cline, M. and Lomow, G., *C++ FAQs*, Addison-Wesley, 1994.
6. Crockford, D., *JavaScript: The Good Parts*, O'Reilly, 2008, Chapter 5.