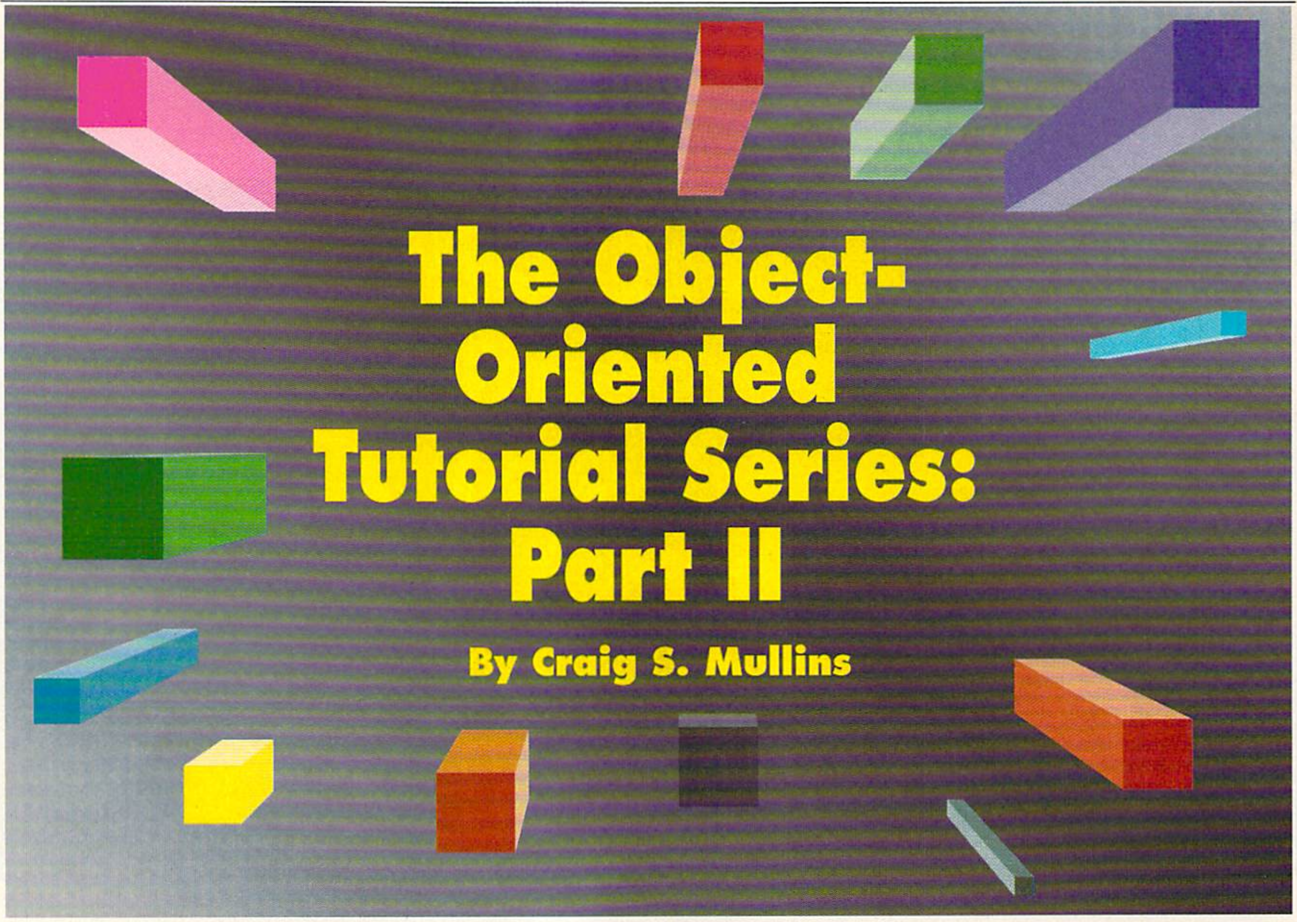


# The Object-Oriented Tutorial Series: Part II

By Craig S. Mullins

In the February issue of *Data Management Review*, we explored the perimeters of the object-oriented (OO) world. The information in that article provided a road map to basic object-oriented concepts for the data base professional. But there is much more to learn. Let's peer deeper into our object-oriented crystal ball to see if we can add some depth to our OO knowledge.

## Where Is That Information Hiding?

Well, I understand the basics. Let's see, objects encapsulate data and methods and are defined by means of a class hierarchy, whereby the data and methods of parent classes are inherited by subordinate classes. Now, what else can you tell me?

Quite a bit, actually. Consider encapsulation. Simply stating encapsulation to be the combination of data and method under one common object is not sufficient. Encapsulation also infers that the data in an object can only be accessed and modified by the methods included therein. This is a key concept. No method can read or update the data in another object. The only means of affecting the data in another object is for a method in one object to pass a message to another object, invoking a method encapsulated in that object.

This concept is called information hiding. When implemented properly, information hiding can increase reusability and decrease software maintenance costs. For any given object, the methods can be completely re-written and the data manipulated without impacting any other part of the system if the object still responded the same way to the same messages.

But, to develop true object-oriented systems, one must change her/his entire way of thinking about software development. Reusability does not just appear because you start using OO techniques. It is incumbent upon the developer to completely plan every phase of the implementation to increase the chance of reusability. Of course, object orientation increases the likelihood that well-planned system development will be reusable. But it simply does not guarantee it. This issue is key to the acceptance of any new technology. There are no panaceas. We must still work hard to produce accurate and usable specifications. Further, we must use standard methodologies to develop the software from the specifications. Failure to do so breeds chaos—and no technology, not even OO, will rescue us from that fate.

More on this later. Let's resume our discussion of OO terminology.

Object orientation also implies polymorphism. Polymorphism can be thought of as almost synonymous to commonality. Let's learn by example. Consider the automobile that you drive each day. What is the driver's interface to that automobile? The gas pedal is long and thin and on the right. The brake pedal is wide and to the left. A steering wheel controls the movement of the car from left to right, and right to left. All of this can be said without knowing anything about your car. This information is generally true regardless of the make, model or year of your car. An automobile, therefore, has a polymorphic interface. The engine within each car that responds to commands from the interface differs by make, model and year, but the actual interface is standard.

But, you may be thinking, American cars have the steering wheel on the right and Japanese cars have the wheel on the left. Doesn't this connote a lack of polymorphism? Not really. When I turn the wheel, it moves the same way regardless of its location within the car. Its location is basically irrelevant.

A lack of polymorphism can be better demonstrated by the switch that controls the high beams of the car's headlights. On some models, this switch is on the floor. It is activated by pressing it with your foot. On other models, the switch is on a lever extending from the steering wheel. This switch is activated by tapping it forward. The different interfaces, each working in a different manner to accomplish the same task, show a lack of polymorphism.

So, objects hide information behind polymorphic interfaces. But how are objects identified? Although the concept is simple, it must be stated. Every object within an OO system or data base must be uniquely identifiable. This should be accomplished by means of an object identifier (OID). It is not at all clear within the OO world as to whether the OID should be system-generated or not. It is also unclear as to whether a primary key can serve the purpose of an OID. Suffice it to say that the objects within an ODB must be uniquely identifiable within the system. The actual make-up and construction of the OID is left to each individual object-oriented data base management system (OODBMS) implementation.

However, as an aside, the availability of a system-generated unique identifier can be of great assistance. Think back to the last time you implemented a data base and one of the tables (or segments) had no easily identifiable primary key. Did you go without one? Did you generate a fake one programmatically? If it was DB2, did you use a timestamp and cause users to deal with its difficult 26-byte character representation? Wouldn't it have been nice for the DBMS to handle it for you? Many OODBMS implementations will automatically generate OIDs.

## Stick Around Awhile

I like it. Information hiding reduces the amount of work that the application developer must do. Polymorphism decreases the learning curve for both users and developers by invoking similar actions the same way. And, every distinct object must be distinctly accessible. It makes perfect sense. But, can you help me with another OO term that I've heard in the trade publications? Just what is "persistence?"

When you are persistent with someone or about something, you simply don't give up and won't go away. Object persistence is basically the same thing

But let's back up and learn by example. Consider the variables within a classic program, such as one written in COBOL. As the program executes, the variables take on values. When the program stops, the variables go away. They are not persistent. To keep the values in between program runs, they must be stored somewhere such as in a flat file, a data base, a report, etc.

If you understand this concept, you understand persistence. For ODBs, objects are persistent. They are stored. For OO programs, the entire state of the program can be persistent. In other words, when an OO program terminates, the state of all of its variables can be saved until the next program execution. So, persistence is just another way of saying that the state of the data is stored somewhere.

Another term that you may have heard in conjunction with object orientation is overloading. This is another simple concept that makes a lot of sense in practice. Simply stated, overloading is the assignment of different meanings to the same method name. The actual process that is performed by the method differs by object. For example, consider the programming language, BASIC. Most implementations of BASIC overload the "+" operator. When two numbers are operated upon by "+", addition is performed. However, when two character strings are operated upon by "+", the strings are concatenated. Two very different operations are performed by the "+" operator depending upon the type of variable. Although the context of this example is not object-oriented, it does provide a good introduction to the concept of overloading.

## Inherit the Wind... And...

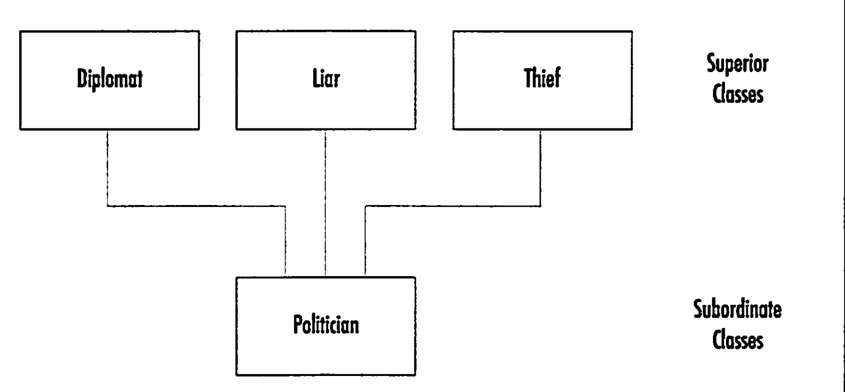
OK, OK, everything so far seems simple. I've heard that OO can get very complex. It seems that if you just take some time to understand the terminology, the haze just lifts. Is it really this easy?

Yes and no. It is true that you must understand the OO jargon before you can understand the OO concepts. But OO can be complex, too. For example, have you heard about multiple inheritance?

Recall from the previous article in this series that inheritance is the technique whereby variables and methods from higher level classes within a class hierarchy are available to be used by lower level classes. But what if a lower level class inherits from more than one higher level class? Furthermore, what if some of these higher level classes incorporate some of the same methods and variables? Consider the example shown in Figure 1.

In the example, the Politician class is subordinate to the following classes: Diplomat, Liar and Thief. The subordinate class is often called the subclass; the superior class is usually called the superclass. If the method "Speak" is contained in both the Diplomat class and the Liar class, which one should the Politician class inherit? Both? Either? Neither? This question is still being debated within the

**Figure 1: Multiple Inheritance**



## OO Terminology

Abstract Data Types — A data type that is not built into the system but is created by the developer to define a specific object class.

Information Hiding — The technique of encapsulating data and processes within a single object to hide the specifics of the implementation from methods that desire to access the data or methods that act upon the data. The implementation of information hiding increases reusability.

Multiple Inheritance — When one class within a class hierarchy has more than one parent class (or superclass).

Overloading — When one method name performs different operations for different objects.

Persistence — The storage of the state of data variables between accesses.

Polymorphism — The use of a common interface to access different implementations of the same basic function or operation.

Subclass — Superior, or parent class, within a class hierarchy.

Superclass — Subordinate, or child class, within a class hierarchy.

OO community. (Although in our somewhat humorous example the answer is probably obvious!) Nevertheless, multiple inheritance is a part of object-oriented technology. Think of the ramifications multiple inheritance could have on data base design! Instead of implementing numerous data stores with numerous procedures operating on the data, ODBs with multiple inheritance could reduce not only the amount of code operating upon the data base structures but also, potentially, the number of data stores.

And the complexity doesn't end there. All MIS professionals understand the concept of a data type. There are several basic data types: integer, decimal and character string. Some languages and data bases may also provide date, time and graphic data types. Object orientation introduces abstract data types. An abstract data type can be any form of data or combination of other data types defined to the OO system. They are not built into the ODB or OOPL, but are typically defined to correspond to an object. Abstract data types can be thought of as the defining structure of a class.

## If I Had 20 Cents for All the Paradigms

So all of my data processing woes can be cured by object orientation, huh? Now where have I heard that before? Maybe when I decided to implement my first relational DBMS. Or was it when I bought into CASE or JAD or Information Engineering, or maybe even way back when structured programming was first introduced. When will one of these new paradigms deliver what I really need—reduced application development time and, even more importantly, reduced maintenance costs?

Well, OO is not magic. Discipline is still required to glean the benefits of OO. But that same statement applies to all of the aforementioned concepts. The failure of any new technology or technique is often times not the fault of the technology but of the marketing hype. Until we in the MIS industry understand that the real cost of developing software lies in developing sound specifications and rigorous models before beginning development, we will never achieve the gains that are desired. And we will never see any new advance as all that it is hyped up to be.

It is true that technology can help to reduce the strain of the software development life cycle (SDLC). But it is just as true that simply implementing new technology without an understanding of the process that it is meant to automate is probably worse than no technology at all. The key to achieving gains in the SDLC with OO is understanding that software development must be planned and modeled before implementation can begin. With clear specifications, OO software and data base development and maintenance can be easier than classic approaches. This is due to the following three factors:

- a closer correlation of the business problem to the computerized implementation;
- an increase in reusability of methods; and
- a decrease in the cost of maintenance due to reduced duplication of code and standard methods.

So the motivation for incorporating OO technology into the SDLC is to achieve more efficient software development. But what is the cost? As with any paradigm shift, the costs can be substantial. Each of the following factors add to the cost of implementing OO techniques:

- lost time and effort due to rejection of the new technology by those who know the old technologies;
- as the realization settles in that object orientation can produce real gains, the cost shifts to time lost as people struggle to shift their knowledge from the classic framework system development paradigm to the OO paradigm;
- depending upon your shop's current acceptance of modeling and SDLC methodology, the cost of introducing these concepts into your shop must be factored in as well if OO is to be successful; and
- as with any new technology, the cost of OO training and education.

## Synopsis

Object orientation provides many benefits over classic software development. But is there an OO revolution on the horizon? Can the benefits of OO be realized simply by modifying relational data base systems with the best OO techniques? Is this possible, and if so, is it even desirable? Part III of this OO tutorial series will compare relational DBMS systems to OODBMS systems. But in the meantime, study the first two installments of this series. Use the sidebar to learn the new terms that we introduced in this article and, as always, keep reading *Data Management Review* for the most up-to-date information on OODBMS.

*Craig S. Mullins is a member of the Technical Advisory Group at* PLATINUM technology, inc. *He has more than seven years of experience in all facets of data base systems development, including developing and teaching DB2 classes, systems analysis and design, data base and system administration and data analysis. You can contact the author via Prodigy [WHNX44A] or Compuserve [70410,237].*

*Was this article of value to you? If so, please let us know by circling Reader Service No. 36.*