A Basic Introduction to OO Concepts

by Rodney Anderson

Object-oriented programming (OOP) is a powerful methodology to improve the processes of design, development, and maintenance of application software. Object-oriented computer languages such as Smalltalk and C++ are used to implement and develop software applications using the object-oriented paradigm.

This article describes just what the object-oriented paradigm is, and uses simple ideas to introduce the concepts of object-oriented programming. These concepts are sometimes difficult to grasp but with perseverance and determination the OO paradigm is understandable and usable. Once a programmer understands OO, there are many worthwhile and rewarding programming discoveries to make. If you are a beginner approaching this subject, read the range of literature available on it. Every article will add a new piece of information to the jigsaw puzzle of understanding OO.

This is intended to introduce you to the main concepts of OO and highlight the benefits of it. Future articles will examine how OOP relates to MWAPI (M Windowing Application Program Interface), to M database techniques, and how OOP can produce a better M development environment.

Introduction

Many people describe the OO paradigm simply as a sophisticated modular library system, but it is much more than that. (M could be described as nothing but a sophisticated text-manipulative language, but it is much more than that.) Just as we may describe the $ORDER and $QUERY functions as being among the most powerful features of M, similarly, the features of OOP have distinct advantages over regular library systems and traditional programming styles.

A good way to approach OOP is to forget all of the computer-programming languages that you know already. Try, instead, to think of everything in the world as a thing or an object. For example, a desk is an object, as is a car, pen, and so on. Often it is easier and quicker for people without computer experience to grasp the concept of OOP than for those who have been programming for many years. To learn about OOP, the best approach is one without preconceived computing ideas.

For my purposes, I will stay away from thinking in computer code or computer languages. Rather, I will describe the OOP in the abstract and consider concepts and ideas in an abstract way, too.

The Principal Concepts

First are the objects. An object can be anything that is visible or tangible or anything that may be perceived intellectually. A knife is a visible, tangible object; somebody's favorite color is an object that is perceived intellectually. Specific objects are referred to as instances. For example, a specific writing desk is an instance, while the color of Fred's hair is an instance of an intangible object.

If many instances can be grouped together in similar types, then this grouping is called a class. John's video, Julie's video, and Tom's video are all instances of the class *video*. A class identifies the classification of instances and defines what these instances will look like and how they will behave. In OOP, instances and classes are structures within the memory of the computer.

Messaging and Methods

Messaging is the act of sending messages to objects. Usually the message is requesting the object to do something. The request might be to obtain data or to update data. An object can respond to a message by executing some computer code. Any computer code executed is commonly referred to as a method.

Messaging is not concerned with what the instances are doing. Sending a message to an instance does not control how the instance responds, nor what the instance does, or even if the instance simply ignores the message. For example, if you have three instances, such as *myCar*, *hisBulldozer*, and *herTrain*, then *myCar* has a steering wheel to control direction, *hisBulldozer* has levers to control direction, and *herTrain* follows the railway tracks. If I send the message "turn left" to each instance, what happens? The instance *myCar* can perform a method "turn left" and will probably turn the steering wheel to the left. Likewise, *hisBulldozer* will execute a method "turn left," which will pull a lever (causing the bulldozer to turn). Both of these are different from *herTrain*, which cannot turn left (or perhaps has no "turn left" method defined) and so will ignore the message.

Polymorphism

The above example about *myCar*, *hisBulldozer*, and *herTrain* also illustrates the concept of polymorphism. Polymorphism means that a method of an instance can be different from a method of the same name in another instance. That is, one instance may respond to a message differently than another, even if the instances are of the same type. The message is the same but the contents of the method (the code) executed may be different. As in the example, the instances respond differently even though they are all one type of vehicle.

Messaging and polymorphism are extremely powerful because they modularize the functionality of the system. A message can say to an instance, "This is the message, I do not care what you do or how you do it, just do it." This also leads to the ability to maintain an OO system easily, because methods are small, debugged procedures that are reusable. To add another instance to the system, I simply define the instance, then define how the instance will respond to the messages that will be sent to it.

This can be illustrated with the above example of three instances of *myCar*, *hisBulldozer*, and *herTrain*. If I want to add a new instance called *yourAirplane* to the class of vehicles, I simply define *yourAirplane* and tell it how to respond to messages. For the "turn left" message, I would describe how *yourAirplane* could turn itself left, and *yourAirplane* could then be used in part of the OO system. The OO system would begin to send *yourAirplane* messages and *yourAirplane* would respond appropriately.

Encapsulation

Encapsulation describes the accessibility of the information within the instance. An instance encapsulates all the methods and data variables within itself. This is one of the more important distinctions between OOP and traditional procedural programming using sophisticated modular library systems. An instance is an independent item that, in a sense, has a "life of its own," separate from other instances in the environment in which it lives. Access to the data of the instance is permitted only by the methods of the instance.

An instance contains all that is needed to make it function and carry out its tasks. This includes procedures needed to respond to incoming messages. The attributes and data that describe the instance are referenced by these methods.

In traditional programming, procedures are merely sets of instructions, possibly with their own local data, which were created for convenience and reuse. The procedures are combined or split, depending on the programmer's personal preference. In OOP, an instance is insulated from the rest of the world. Since the instance executes only internal procedures and uses information about itself, information contained within it, or parameters passed to it, it is much easier to track down problems and less likely that a bug in one instance will affect some other part of the system in an apparently unconnected way.

When an OOP system is running live, the data in the object will not change between one use of the instance and the next, unless a method or message is received to change it. Instances can remember data values since variables are not initialized automatically. An object's internal variables are often referred to as properties, attributes, or instance variables. These terms are often mixed up. It's important to note that in OOP, messaging is the only way to get the value of the property of an instance. Access to the attributes of an instance is made only by sending messages to the instance.

An instance can be said to have a state. The state of an instance is important and consists of the current values of its variables. That is, the instance's information about itself makes up the state of the instance.

Classes and Instances

A class is a template for an instance. A class defines how an instance will look and behave.

The instances *hisBulldozer*, *herTrain*, and *yourAirplane* clearly are instances, but where do they come from? It is possible to create all these instances individually. Begin with nothing and uniquely create each object, but this is not always practical. What we need is a factory that defines how the instance will look and behave, and when requested will produce a new instance of the item. This factory is called a class.

We need a Car class to produce cars. The Car class defines how the car looks and behaves. When requested, the Car class can produce a new instance of a car. Similarly, we need a Truck class to produce trailer trucks. The Truck class defines how the truck looks and how it behaves. When requested the Truck class can produce a new instance of a truck.

Base classes usually are predefined and supplied in OOP languages. New classes, however, may be required and created. A class also has the following functionality:

- Classes group similar instances together into a classification. That is, a group of cars could belong to the Car class.

- Classes are used to produce new objects. Similar to a factory, a class called Ford Cars not only classifies and describes Ford cars, but can produce a new *fordcar* instance.

An instance is a unique object with a unique identifier with characteristics described by the class. Obviously, a class may produce many instances. An instance is created when a class receives the message NEW. The class will produce the new instance, initiate the properties and methods of the instance and return the identifier of the instance. Instances have a defined lifetime and can be easily created and destroyed.

Another way to consider classes is to think of them as factories. If you send a message NEW to the factory then it produces a new instance of the class. If the factory described and manufactured cars, then when the factory (class) received a NEW message, it would produce (create) a new instance of a car and give the location of the new car.

The name of the factory could be *Car* while the instance of the new car could be identified as *myCar*. Many OO systems and many OOP books distinguish between a class and an instance by using an uppercase letter for a class (*Car*) and a lowercase letter for an instance (*newCar*).

Inheritance

Inheritance is another extremely powerful concept of OOP. As the name suggests, an instance may be a child of another instance and have the same properties as the parent instance. This occurs in the same way a human child may inherit the same eye color from its father. A child may inherit its eye color from its parent, which means that the child could have the same eye color as the parent. But the actual eye color may be different from the parent. It can be said, then, that the eye-color property is always inherited (meaning that all children have colored eyes) but the actual value may be inherited (the same as the parent) or not inherited (different from the parent). For example, *myCar*, *hisTruck*, and *herBus* have properties. Some of them are common to all three instances and some are not. They all have an engine, a driver seat, and wheels as common properties, yet they usually have different vehicle bodies (uncommon property).

So we could create a class of these instances called vehicle and describe the common properties of the instances here. The vehicle class would be the parent of the three instances. That is, the vehicle class describes the engine, the driver seat, and the wheels. The instance would inherit these properties from the vehicle common parent class. This is also called a base class. The body of each instance, however, would be unique and would be described in each instance.

The instance *myCar* would have a car body; *hisTruck* would have a flatbed (to carry large loads), and *herBus* would have many passenger seats in its body.

Methods also could be common to the instances and usually inherited from the base class. For example, our three instances would have different "turn left" methods but possibly the same "go forward" method. Properties also can be inherited from the base class and become the default properties of the child instance.

Since an instance can inherit properties and methods from its parent class, we need to qualify our encapsulation concept. An instance contains at run-time all that is needed to make it function and carry out its tasks. OOP languages implement late binding of inherited methods and properties to the instance. In OOP, this late binding occurs at run-time.

Class Hierarchy

Object-orientation always starts with one basic class. We can call this class OBJECT. The next stage in building a class hierarchy is to define a new class, called a derived class, which is a child of the base class. The derived class will be a child of the base class and inherit the methods and properties of the parent class. The properties and methods are usually not copied to the derived class but stored in the base class. The derived class may include additional methods and properties that did not exist in the base class; it also may redefine methods (or even scrap them altogether). It also may assign new values to properties, which are then stored at the derived class. A derived class may disinherit properties or methods and therefore have fewer characteristics than its base class.

Life in OOP is simple, since a derived class does not have to redefine all the methods or properties in the base class. Instead, when a programmer declares that a new class is derived from a base class, only those properties and methods that are new or changed require definition. All other properties and methods from the base class are assumed to be inherited by the derived class as well. This has the great advantage that when a property or method is altered in a base class, the same change automatically applies to all derived classes (unless the property or method is redefined in a derived class).

The process of inheritance can (and usually will) continue over a series of classes. A class that is derived from a base class can itself become the base class for other derived classes. In this way, object-oriented programs build up a class hierarchy. It is important to note that a class hierarchy cannot be circular. This means that a class cannot become a parent to one of its ancestors.

The terminology I have used was chosen carefully to avoid confusion. Since the word *object* is generic and ambiguous, I have avoided using this word whenever possible. Two types of objects, however, have been identified and labelled. The first type of object is an instance object, which is a specific instance of a class and uses a lower-case letter, e.g., ford car.

Summary

The initial concepts of the OOP have been presented here in simple terms. These concepts include:

- The behavior of an object is determined by its class. A class is a template for objects, each new object being an instance of the class.
- Instances from different classes may have methods with the same names, but these methods may respond in different ways. This is polymorphism.
- An instance encapsulates all the methods (procedures and functions) and data needed for it to operate correctly. The instance usually operates through the public methods it inherits from its class. The values of the instance variables determines its state.
- New classes are derived from base classes. Each derived class can be the base class for other classes. The result is a class hierarchy.
- A class inherits the methods of its base class. This base class, in turn, inherits methods from its base class. (This means that a class inherits methods from all classes above it in the hierarchy.) A class can add new methods or override those of the base class by redefining them. Circular hierarchy is not allowed.

OOP has many features that make it perfect to add to the M programming development environment. Employing the OOP approach with other programming techniques (Case-tools, SQL, and so forth) makes application development easier and faster. **M**

Rodney Anderson is Australian and has worked with M since 1980. He worked in the UK and Germany with Micronetics Design Corp. during 1992 and 1993 on graphical user interface. Since then, he has been developing MWAPI OO tools. Write to him at P.O. Box 1633, Macquarie Centre, NSW 2113, Australia, or fax him at 61-49-527878.

$$\begin{tikzcd}
\boxed{\text{MIIS}^{\text{TM}}} \arrow[d] \\
\boxed{\text{POLYLOGICS}} \arrow[d] \\
\boxed{\text{MUMPS}}
\end{tikzcd}$$

We turn running MIIS programs into running MUMPS programs. Efficiently, with maximum accuracy and minimum down-time.

MIIS in, MUMPS out. That's all there is to it.

We specialize in MUMPS language conversions. We also convert MAXI MUMPS, old MIIS, BASIC and almost anything else into standard MUMPS. Polylogics will be there with experienced project management, training and documentation.

So, give us a call today. Ask for a free demonstration on a few of your programs. That's all there is to it.

POLYLOGICS CONSULTING

136 Essex Street
Hackensack, New Jersey 07601

Phone (201) 489-4200
Fax (201) 489-4340

MIIS is a trademark of Medical Information Technology, Inc.