



On the Notion of Object-Oriented Programming

This article discusses the fundamentals of object orientation, and in the process dispels some myths about it.



< OOP >

Object-oriented programming (OOP) has been the most popular programming paradigm for more than two decades and has been the subject of active research for quite some time now.

With such widespread use, we would expect the essence or basic tenets of OOP to be widely known to students and practitioners. However, the essentials are not often understood or followed well in practice. By just programming in an object-oriented language does not necessarily mean that we do that task well. In this article we will look at the fundamentals of object orientation and try to understand why it has become so successful. We'll also dispel some myths and face some hard facts about OOP technology.

The basics

What is object-oriented programming? Hanspeter Mossenbock's *Object-Oriented Programming in Oberon-2* [Springer-Verlag, 1993] provides the following definition: "Object-oriented programming means programming with abstract data types (classes) using inheritance and dynamic binding." This makes it clear that OOP is about using ADTs with inheritance and virtual functions.

Many of the attributes we often associate with OOP aren't inherently associated with this definition. Strictly speaking, having a common base class (say, the object class), or language features like namespaces, exception handling, RTTI (RunTime Type Identification), reflection, operator overloading, function overloading, etc. are nothing to do with OOP, as such. However, many modern OOP languages (like Java) have a lot of these features, which doesn't mean they are characteristics of object orientation itself. A language can have none of

these features and still can be considered as a good OO language.

An example of flawed arguments

I recently heard the following argument: "Java is better than C++ because Java is a 'pure' object-oriented language. Java allows no global variables, it has a common Object base class, and it does not support multiple inheritance and so on. But C++ is a bad object-oriented language because it supports procedural programs, has no common base class, supports multiple inheritance, etc."

I like both Java and C++, but that aside, let us analyse why this argument is inherently flawed.

Just because we believe that Java is a 'pure' OO language, it need not be better than C++. Why is 'pure' object-orientation better than any other approach (like the multi-paradigm approach in C++)? What are the criteria for concluding whether a language is 'pure' or 'impure'? Doesn't Java's support of C-style conditional and looping constructs, and primitive types make it 'impure'? Just because a language also supports other paradigms doesn't mean that it is a bad OO language.

Providing a common base class is a design decision that a language designer makes in an OO language. Based on information and category theories, many researchers have supported the argument that having a common class as a base class for all the classes is unnatural and a non-intuitive design.

Inheritance is one of the fundamental features of OOP. A language without any form of inheritance support cannot claim to be an OOP language. Multiple inheritance is just one kind of inheritance and it is common to see it in use (for example, a two-in-one set inherits the properties of both a tape-recorder and a radio). Practically, it is difficult to use multiple inheritance correctly, and

the problems that may need multiple inheritance can be rewritten using only single inheritance (possibly with some difficulty). Because of such reasons, a language designer can make a design decision to support or not support multiple inheritance in her language. However, just because an OO language supports multiple inheritance we cannot label it as a 'bad' language.

As Paul Weiss wisely commented: "It's one thing not to see the forest for the trees, but then to go on to deny the reality of the forest is a more serious matter."

Characteristics and benefits

OOP is based on the solid foundation provided by the concepts of inheritance, polymorphism, abstraction and encapsulation.

An abstract data type provides us with the ability to think of a type in its abstract interface level and use it depending on its interface only (without concerning ourselves with the implementation details; otherwise, there is nothing 'abstract' about an ADT). Since OOP is based on the solid foundation provided by abstraction, it is possible to create complex software that hides a lot of implementation details. Because of this, object orientation is suitable for programming on a large scale.

Inheritance is the mechanism in which common properties of various classes (or objects, in case of object inheritance) are abstracted and provided in common base classes, thereby creating relationships between related types. Inheritance is important for design since it helps to organise classes in terms of the natural relationship between the types. In inheritance many of the details of the derived classes will only be known later in the design stage. Also, because of the

classification of related types, we can write code for general types (instead of code that works or assumes specific types). This gives rise to runtime polymorphism where late binding of actual methods for the code is resolved. With inheritance (and runtime polymorphism), it is easier to introduce new changes and extend the software with new types without affecting the rest of the code, so OOP programming enables writing extensible software. Object orientation also enables providing abstract or concrete classes as libraries and related classes as frameworks; instead of (re)writing classes every time, we can reuse the classes. So object orientation helps writing reusable software.

So the ability to handle complex code (enhanced maintainability), the ability to extend the code based on specific requirements later (extensibility) and the ability to use or reuse the code (reusability) are the main benefits promised by OOP.

Costs and drawbacks

OOP is not a panacea for the problems faced by the software industry today. With more than three decades of experience in using object-oriented programs, the hype and over-enthusiasm in promoting object orientation has subsided and the costs of object-oriented programming are clear.

Object orientation requires a new way of thinking (different from procedural thinking), and it is difficult to comprehend object-oriented programs. Since object orientation provides a higher level of abstraction from machine-level details, there is some loss of efficiency (compared to procedural programs).

The benefits of OOP aren't automatically available to programs written in the procedural way—by just using OOP constructs. For programmers with a procedural

background, it takes more time to learn the object-oriented approach than to learn the language features.

Object-oriented programs are typically more difficult to understand than their procedural equivalent. For example, to understand how to use a derived class, we need to go through the public members of that class and also the public members of all of its base classes. The effort required to understand and make use of class libraries, frameworks or inheritance hierarchies is considerable.

Object orientation provides a higher level of abstraction from low-level details and it provides features to model higher-level relationships between classes. This flexibility also comes with some loss of efficiency compared to straightforward procedural constructs. For example, to support runtime polymorphism, an extra level of indirection is needed to invoke a virtual method. In other words, a virtual method call is almost always slower than a direct method call. Such additional overhead for abstracting low-level details is known as an abstraction penalty in OOP.

Some myths and realities

We will encounter many myths associated with OOP. Let us check the validity of some of them here.

Software written in OOP is better than non-OOP software.

False! While it is true that OOP provides significant help in creating quality software, the quality of the software we create is in our hands, and good or bad software has nothing to do with OOP. We can write high-quality code in other programming paradigms (like functional programming); similarly, we can also write sloppy code following OO approach. So, it is incorrect to make claims like: "My software is better than your's because mine is OO whereas yours isn't!"


With OOP we can write generic, reusable components and sell it independently.

Partially true! One of the fundamental benefits with OOP is the ability to write reusable code, but OOP hasn't delivered fully in the writing and distributing of generic reusable code, in practice. Writing COTS (Components Off The Shelf) software is still not fully possible with OOP. The emerging areas of component software and generic programming paradigms address this shortcoming of OOP.

OOP emerged based on solid theoretical foundations.

False! Unlike programming paradigms like functional programming, which is based on lambda calculus and emerged from extensive academic research, object orientation came into practical use first and was later subjected to extensive research. Theoretical understanding of many of the areas on OOP is still hazy and OOP continues to be a subject of active research.

It takes time to master object-oriented programming and design.

True! It is easy to understand the basic tenets, features and ideas of OOP. However, it takes considerable time to master the analysis, design and implementation techniques to create quality solutions using object-oriented technology. As they say, nothing good ever comes easy! 

By: S.G. Ganesh is a research engineer in Siemens (Corporate Technology). His latest book is "60 Tips for Object Oriented Programming", published by Tata-McGraw Hill in December this year. You can reach him at sgganesh@gmail.com