Lecture 09: Support for Object-Oriented Programming

This lecture discusses how programming languages support object-oriented programming. Topics to be covered include:

- Introduction
- Fundamental Principles
- Design Issues
- Cases: Support in Specific Languages
- Summary and Concluding Remarks

9.1 Introduction

You are likely aware that object-oriented programming (OOP) has been dominating contemporary programming, and for good reasons. Among the fundamental tenets of OOP are the following: classes, subprogram overloading, operator overloading, inheritance, polymorphism, amalgamation, and dynamic binding.

Class definition was discussed in the previous lecture. In most object-oriented programming languages (OOPLs), classes are specified by using the keyword **class** as part of the syntax. On exception to this is Ada, where classes are implemented as specification packages (via the keyword **package**) and body packages (via keywords **body package**). The body package defines implementation of the items in the specification package.

As pointed out in lecture 8, each OOPL tends to have its own idiosyncrasies when it comes to class definition and implementation. Rather than focusing on that, the rest of this lecture will focus on how some of the other fundamental tenets are implemented in selected OOPLs and/or hybrid languages.

9.2 Fundamental Principles

In looking at the fundamental principles of OOP (classes, subprogram overloading, operator overloading, inheritance, polymorphism, amalgamation, and dynamic binding), it can be observed that there is widespread acceptance of these principles across various languages; however, implementation details vary from one language to another. Following is a brief survey.

9.2.1 Subprogram Overloading

Subprogram overloading is the ability to define a subprogram multiple times, each with a slightly different header definition and corresponding code. The compiler invokes the appropriate subprogram based on the context of usage.

This feature is possible only in languages that support dynamic binding. It is supported in languages such as C++, Java, and C#.

9.2.2 Operator Overloading

Operator overloading is the ability to redefine the meaning and application of an operator. Most language support operator overloading at the basic level. For example, Java allows the + operator to be applicable to numeric data as well as strings.

Languages such as C++, Python, and Ruby allow the programmer to also overload operators to meet specific needs. C++ takes operator overloading to another level by allowing the programmer to redefine the meaning and application of almost all its operators. For more on this, review lecture 7 as well as any credible C++ resource center.

9.2.3 Inheritance

Inheritance is the capacity of an object to adopt the features of its parent class on which it has been defined. It also relates to a class adopting the features of another class. The principle can be traced back to SUMULA 67, but did not really gain momentum until the 1980s.

All OOPLs support inheritance but not necessarily in identical ways. Below is a summary of three cases:

- **C++:** The language supports single inheritance as well as multiple inheritances. It uses a somewhat cryptic notation. Keywords supported are **public**, **private**, and **protected**. Private properties are not inheritable.
- **Java:** Java supports single inheritance, but does not support multiple inheritances. However, through interfaces and abstract classes, limited benefits of multiple inheritances can be achieved, while eliminating the drawbacks. Java can block a method from being inheritable via allowing use of the keyword **final**. Other keywords supported are **public**, **private**, and **protected**. Private properties are not inheritable.
- **C#:** This language supports inheritance in exactly the same way as C++, and with syntax identical to that of C++.

9.2.4 Polymorphism

Polymorphism is the capacity of an object, operation, or operator to take on different forms, depending on the context of usage. This is achieved through subprogram overloading, operator overloading, subprogram with default arguments, and subprogram overriding, generic subprograms, object inheritance, and object casting.

By way of review, subprogram overriding may proceed in any of the following ways:

- A subclass may inherit methods from its super-class, and adopt them without changes.
- A subclass may inherit a method and modify the original behavior to behave differently.
- A subclass may inherit a method and restrict its original behavior.
- A subclass may inherit a method and extend the original behavior to include other instructions.

All OOPLs support polymorphism to different degrees. Below is a summary:

- **Java** supports method overriding, method overloading, and limited operator overloading (programmers are not allowed to overload operators). Method overriding also extends to abstract classes and interfaces. Generic subprograms are supported in a limited way.
- **C++** supports function overloading, function overriding, operator overloading (including the ability to redefine the meaning and application of operators), and function with default arguments, generic subprograms, and object casting.
- **C#** supports function overloading in a manner that is similar (and in many respects identical) to how it is done in C++.

9.2.5 Dynamic Binding

Dynamic (late) binding is a characteristic feature of all OOPLs. Without it, it would be difficult to support other principles of inheritance and polymorphism.

9.2.6 Amalgamation

Amalgamation is also characteristic feature of all OOPLs. Through composition and/or aggregation, complex objects can be constructed with minimum effort. This feature is facilitated in all OOPLs.

9.3 Design Issues

When an OOPL or hybrid language is being designed, the software engineers and language designers have to carefully consider what OOP tenets will be supported in the language, to what extent, and how. These decisions have to be made early in the design phase of the software development life cycle (SDLC), since they tend to affect other aspects of the language. Do note and understand that designing a programming language is a huge software engineering undertaking that requires various skills. Following are some of the main considerations.

Object Exclusivity: Should everything be implemented as objects, or should there be exceptions?

Subclasses vs Subtypes: Should subclasses always be subtypes? Is the **is-a** relationship always applicable for subclasses?

Abstract Classes: As you are aware, an abstract class is a class for which instances are not created. Abstract classes are purely for the purpose of inheritance. Will they be supported, and if yes, how?

Type Checking & Polymorphism: To what level should polymorphism be taken, and how does this affect type checking? In most cases, casting solves this problem.

Single vs Multiple-Inheritance: Should multiple-inheritance be supported? If yes, how will the negative side-effects be avoided?

Object Allocation & De-allocation: Know how this is done in different languages. One approach is to treat objects as ADT instances. This means they can be allocated from the program's run-time stack, or explicitly put on the program's heap via an operation such as **new**. The latter strategy provides more flexibility: heap nodes are object references, while stack nodes tend to be value variables.

Another issue is whether objects are de-allocated implicitly or explicitly. Implicit de-allocation requires some internal method of storage recapture; explicit de-allocation raises concerns about treatment of dangling pointers.

Binding Strategy: Late binding versus early binding — which strategy will be employed?

Object Initialization: Will object instantiation be implicit or explicit? Or will the language support both strategies? CS professionals studying programming languages should know how this is done in different languages.

Nested Classes: Some languages support nested classes (Java is one though it is not encouraged); others do not (for example C++).

9.4 Cases: Support in Specific Languages

Let us look at how the OOP tenets of section 9.1 are met in specific programming languages. We will look at five languages (Smalltalk, C++, Java, C#, and Ruby), but you are encouraged to look at others not covered in this lecture.

9.4.1 Smalltalk

- Purely OOPL
- **Inheritance:** Superclass properties are all inherited in the sub-class by default; none can be hidden; single inheritance supported
- **Dynamic binding** supported
- **Performance:** Slow

9.4.2 C++

C++ is a hybrid language — supporting both imperative (procedural) and OO programming. C++ supports both classes and structures; the former has member functions while the latter typically does not.

Inheritance:
- Multiple-inheritance is supported; the virtual keyword avoids ambiguity.
- The parent constructor for a sub-class object is automatically called at instantiation
- Private properties cannot be inherited
- Scope resolution operator is used to access a class property beyond the boundaries of the class

Polymorphism:
- Supports subprogram overloading and overriding
- Supports operator overloading more comprehensively than most other OOPLs
- Supports generic subprograms and classes through templates

Binding: Late binding is supported.

Performance: The language is very efficient and portable language.

9.4.3 Java

Java is accepted as an OOPL but is technically a hybrid language — supporting both imperative (procedural) and OO programming. Java supports classes only; structures are not supported.

Inheritance:
- Single inheritance is supported; multiple inheritance is supported
- Instead of multiple-inheritance, Java supports interfaces. A Java class may implement multiple interfaces
- Java also supports abstract classes — classes for which instances are not directly created, but which can be inherited in an inheritance hierarchy.
- The parent constructor for a sub-class object is not automatically called at instantiation
- Private properties cannot be inherited
- The **super** keyword is used to access a class property beyond the boundaries of the class to the parent class

9.4.3 Java (continued)

Polymorphism:
- Java supports subprogram overloading and overriding
- Java provides operator overloading in a very limited way with respect to the + operator. However, programmers are not allowed to overload operators.
- Java supports generic subprograms and classes

Binding: Late binding is supported.

Performance: Java is a very portable language; through the Java Virtual Machine (JVM), platform independence is achieved.

9.4.4 C#

C# is accepted as an OOPL but is technically a hybrid language — supporting both imperative (procedural) and OO programming. C# supports both classes and structures; the former has member functions while the latter typically does not.

Inheritance:
- Single inheritance is supported; multiple inheritance is supported
- C# also supports abstract classes — classes for which instances are not directly created, but which can be inherited in an inheritance hierarchy.
- The parent constructor for a sub-class object is not automatically called at instantiation
- Private properties cannot be inherited
- The **super** keyword is used to access a class property beyond the boundaries of the class to the parent class

Polymorphism:
- C# supports subprogram overloading and overriding
- C# supports operator overloading in a manner that is similar to C++
- C# supports generic subprograms and classes

Binding: Late binding is supported.

Performance: C# enjoys wide support in the .NET family of software development tools.

9.4.5 Ruby

Ruby is a pure OOPL: virtually everything is an object and computations are done via message passing. Like Java, Ruby supports classes only; structures are not supported. Data items of a class are by definition private in nature, and are not allowed to be anything else. Constructors have a standard name *initialize*.

9.4.5 Ruby (continued)

Inheritance:
- Single inheritance is supported; multiple inheritance is supported
- Ruby does not support interfaces.
- Private properties cannot be inherited
- The **super** keyword is used to access a class property beyond the boundaries of the class to the parent class

Polymorphism:
- Ruby does not support subprogram overloading but supports subprogram overriding
- Like C++, Ruby allows the programmer to define how operators are used in a class.
- Since Ruby does not support the concept of predefined types, the concept of generic subprogram is irrelevant.

Binding: Late binding is supported.

Performance: The language is not as efficient as typical compiled languages.

9.5 Summary and Concluding Remarks

Let us summarize the salient points of this lecture:
- The fundamental tenets of OOP include the following: classes, subprogram overloading, operator overloading, inheritance, polymorphism, dynamic binding, and amalgamation. These principles are widely accepted in OOPLs and hybrid languages. However, they are implemented differently across different language barriers.
- When an OOPL or hybrid language is being designed, the following are some design issues to be considered early in the design stage: object exclusivity; subclasses versus subtypes; abstract classes; type checking and polymorphism; single versus multiple-inheritance; object allocation and de-allocation; binding strategy; object initialization; nested classes versus stand-alone classes.
- We have looked at how the aforementioned OOP tenets are met in five specific programming languages (Smalltalk, C++, Java, C#, and Ruby), but you are encouraged to look at others not covered in this lecture.

Like the previous lecture, this is a rather fascinating topic that we could easily continue exploring. You are encouraged to conduct that exploration with various OOPLs and hybrid languages. Our next lecture will change gear a bit and examine the matter of concurrent programming — a rather relevant topic in contemporary programming.

9.6 Recommended Readings

[Foster 2017a] Foster, Elvis C. 2017. *C++ Programming Fundamentals*. Accessed March, 2018. https://www.elcfos.com/lecture-series/index/entry/id/10/title/C++ProgrammingFundamentals

[Foster 2017b] Foster, Elvis C. 2017. *Lecture Notes in Programming Foundations*. Accessed March, 2018. https://www.elcfos.com/lecture-series/index/entry/id/10/title/ProgrammingFoundations

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapters 7 & 9.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 12.