Category: Software & Systems Design

Inheritance in Programming Languages

Krishnaprasad Thirunarayan
Wright State University, USA

INTRODUCTION

Inheritance is a powerful concept employed in computer science, especially in artificial intelligence (AI), object-oriented programming (OOP), and object-oriented databases (OODB). In the field of AI, inheritance has been primarily used as a concise and effective means of representing and reasoning with common-sense knowledge (Thirunarayan, 1995). In programming languages and databases, inheritance has been used for the purpose of sharing data and methods, and for enabling modularity of software (re)use and maintenance (Lakshmanan & Thirunarayan, 1998). In this chapter, we present various design choices for incorporating inheritance into programming languages from an application programmer's perspective. In contrast with the language of mathematics, which is mature and well-understood, the embodiment of object-oriented concepts and constructs in a concrete programming language is neither fixed nor universally accepted. We exhibit programs with similar syntax in different languages that have very different semantics, and different looking programs that are equivalent. We compare and contrast method inheritance, interaction of type system with method binding, constructs for method redefinition, and their implementation in widely used languages such as C++ (Stroustrup, 1997), Java (Arnold, Gosling, & Holmes, 2005), and C# (Hejlsberg, Wiltamuth, & Golde, 2006), to illustrate subtle issues of interest to programmers. Finally, we discuss multiple inheritance briefly.

BACKGROUND

SIMULA introduced the concepts of *object*, *class*, i*nheritance*, and *polymorphism* for describing discrete event simulations (Meyer, 1999). Subsequently, object-oriented programming languages such as Smalltalk, C++, Object Pascal., and so forth used these concepts for general purpose programming (Budd, 2002).

Object/Instance is a run-time structure with state and behavior. Object state is stored in its fields (variables) and behavior as its methods (functions). *Class* is a static description of objects. (In practice, a class itself can appear as a run-time structure manipulated using reflection APIs such as in Java, C#, CLOS, and so forth.) A class defines type of each field and code for each method, which inspects and/or transforms field values. (By field we mean *instance* field, and by method we mean *instance* method. The discussion of static fields and static methods, and access control primitives such as private, protected, and private, are beyond the scope of this chapter.) *Inheritance* is a binary relation between classes (say P and Q) that enables one to define a class (Q) in terms of another class (P) *incrementally,* by adding new fields, adding new methods, or modifying existing methods through overriding. A class Q is a *subclass* of class P if class Q inherits from class P.

```
class P {
      int i;
      int f() { return 2;}
      int f1() { return f() + i;}
      void g() {}
}
class Q extends P {
      int j;
      int f() { return 4;}
      int h() { return i + j;}
}
class Main {
      public static void main(String [] args) {
            Q q = new Q();
            P p = q;
            p.f1();
      }
}
```

Every P-object has an int field i, and methods f(), f1() and g() defined on it. Every Q-object has int fields i and j, and methods f(), f1(), g() and h() defined on it. Q inherits i, f1(), and g() from P, and overrides f() from P.

The variable q of type Q holds a reference to a Q-instance (an object of class Q) created in response to the constructor invocation 'new Q()' (Gosling, Joy, Steele, & Bracha, 2002). The variable p holds a reference to the same Q-instance as variable q (*dynamic aliasing*) through the *polymorphic assignment* 'p = q.' In general., *polymorphism* is the ability of a variable of type T to hold a reference to an instance of class T and its subclasses. The method f1() can be invoked on the variable p because it is of type P and f1() is defined in class P. The method f1() can be successfully invoked on a Q-instance due to method inheritance. *For a subclass to be able to reuse separately compiled method binaries in the ancestor class, the layout of the subclass instances should coincide with the layout of the ancestor instances on com-*

mon fields. The body of f1() invokes f() defined in class Q on a Q-instance referred to by variable p through *dynamic binding.* In other words, it runs the code associated with the object's class Q rather than the variable's type P. *For the method calls compiled into the ancestor's method binaries to be dynamically bound, the index of the method pointers in the ancestor method table and the descendent method table should coincide on the common methods.*

The implementation technique for reusing parent method binaries in a straightforward way works for languages with only single inheritance of classes. A language supports *multiple inheritance* if a class can have multiple parents. The implementation of multiple inheritance that can reuse separately compiled parent method binaries requires sophisticated manipulation of object reference (self/this pointer adjustment) (Stroustrup, 2002, Chapter 15), or hash table based approach (Appel, 2002, Chapter 14), in general.

Object-oriented paradigm and imperative/procedural paradigm can be viewed as two orthogonal ways of organizing heterogeneous data types (data and functions) sharing common behavior. The relative benefits and short comings of the two paradigms can be understood by considering the impact of adding new functionality and new data type. Procedural paradigm incorporates new functions incrementally while it requires major recompilation to accommodate new data types. In contrast, the object-oriented paradigm assimilates new data types smoothly but requires special *Visitor* design pattern to deal with procedure updates. Another significant advantage of object-oriented paradigm is its use of *interface* to decouple clients and servers. Wirth (1988) elucidates type extension that bridges procedural languages such as Pascal to object-oriented languages such as Modula-3 via the intermediate languages such as Modula and Oberon.

COMPARISON OF METHOD INHERITANCE IN C++, JAVA, AND C#

In this section, we discuss subtle issues associated with method inheritance in programming languages using examples from C++, Java and C#.

Single Inheritance and Method Binding in C++ vs Java

Consider a simple class hierarchy consisting of Rectangle, ColoredRectangle, and Square, coded in Java. The state of the instance is formalized in terms of its width and its height, and stored in int fields w and h. The behavior is formalized using perimeter() method which is *defined in* Rectangle, *inherited by* ColoredRectangle, and *redefined/overridden in* Square (let us say for efficiency!).

```
class Rectangle {
    int w, h;
    int perimeter() { return (2*(w+h)); }
}
class ColoredRectangle extends Rectangle {
    int c;
}
class Square extends Rectangle {
    int perimeter() { return (4*w); }
}
class OOPEg {
    public static void main (String[] args) {
        Rectangle [] rs = { new Rectangle(),
            new ColoredRectangle(), new Square()};
        for (int i = 0 ; i < rs.length ; i++ )
            System.out.println( rs[i].perimeter() );
    }
}
```

The array of rectangles is a polymorphic data structure that holds instances that are at least a Rectangle. The for-loop invokes the "correct" perimeter-method on the instance referred to by the polymorphic reference rs[i] through run-time binding of the call rs[i].perimeter() to the method code based on the class of the instance referred to by rs[i] (dynamic type of rs[i]) rather than the declared type of rs[i] (static type of rs[i]).

This code can be minimally massaged into a legal C++ program. (Note that perimeter is explicitly prefixed with keyword *virtual* in C++. #include's have been omitted.)

```
class Rectangle {
    protected int w, h;
    public virtual int perimeter() { return (2*(w+h)); }
}
class ColoredRectangle : public Rectangle {
    private int c;
}
class Square extends Rectangle {
    public int perimeter() { return (4*w); }
}

void main (char* argv, int argc) {
        Rectangle rs [3] = { Rectangle(),
            ColoredRectangle(),  Square()} ;
        for (int i = 0 ; i < RSLEN ; i++ )
            cout << rs[i].perimeter() << endl;
}
```

The main(…)-procedure in C++ resembles the corresponding main()-method in Java syntactically, but they are very different semantically. The array of Rectangle is a homogeneous structure with each element naming a Rectangle instance. The initialization assignments cause the common fields to be copied and the additional subclass instance fields to be ignored (projection). In other words, there is no polymorphism involved. Similarly, the call rs[i].