Inheritance in Programming Languages

Krishnaprasad Thirunarayan

Metadata and Languages Laboratory

D e p a r t m e n t   o f   C o m p u t e r   S c i e n c e   a n d   E n g i n e e r i n g

W r i g h t   S t a t e   U n i v e r s i t y

D a y t o n ,   O H - 4 5 4 3 5

I N T R O D U C T I O N

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 paper, 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 et al, 2005), and C# (Hejlsberg et al, 2006), to illustrate subtle issues of interest to programmers. Finally, we discuss multiple inheritance briefly.

B A C K G R O U N D

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, etc. 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, etc.) 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 paper.) ==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 et al, 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 common 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* <mark>*dynamically bound*</mark>*, 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 as discussed above 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]).

The above 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].perimeter is *statically* bound and invokes the code in class Rectangle on a "direct" instance of Rectangle.

The above driver code can be modified into another C++ program that is *equivalent* to the Java program given earlier using pointers and dereferencing operator.

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

The array of Rectangle is a homogeneous structure of polymorphic references with each element holding a reference to a (possibly indirect) Rectangle instance. The initialization assignments cause the array to hold references to three instances : a Rectangle, a ColoredRectangle and a Square. Similarly, the call rs[i]->perimeter is dynamically bound. In other words, for the ColoredRectangle instance, it runs the inherited code from class Rectangle, while for the Square instance, it runs the redefined/overriding code from class Square.

Java uses dynamic binding of methods as the default, while C++ uses static binding of methods as the default. To specify dynamic binding in C++, an explicit keyword *virtual* is necessary in front of the original overridden method definition.

Method Redefinition in Java and C#

C# resembles C++ and differs from Java in that dynamic binding of methods is not the default. To override a method, the overriding method signature must not only match the overridden method's signature, but it must also contain the keyword *override*. It is also possible to support a new subclass method that matches the parent method's signature but is logically distinct from it using the keyword *new,* instead of *override*. The rationale for this design decision is to achieve robustness in the context of code evolution in the face of changes to the parent class methods.

Consider the following C# class definitions.

```
class Parent {…}
class Child : Parent {   public virtual void m(){…} }
```

A subsequent update to the parent class with a new definition of the method m() as shown below goes undetected in Java, while C# throws an error.

```
class Parent {  public virtual void m(){…}  }
```

In C#, the programmer can state that the method m() in the subclass Child is related to the method m() in the class Parent by changing the former using the keyword *override* or proclaiming their independence using the keyword *new*.

```
class Child : Parent {   public virtual override void m(){…} }
```

```
class Child : Parent {   public virtual new void m(){…} }
```

Method Overriding in C++ and Java

A method defined in a class is guaranteed to be available in all descendant subclasses in Java (signature-based subtyping). A subclass may either inherit the method code from the ancestor or override it. In contrast, this claim does not hold in C++.

The following Java code compiles without any error. Both Child and GrandChild instances have two methods defined on them: one with signature m(int) and another with signature m(int, boolean).

```
class Parent { void m(int i) {  } }
class Child extends Parent { void m(int i, boolean b) {  } }
class GrandChild extends Child { void m(int i, boolean b) {  } }
class Overload {
  public static void main(String[] args) {
    Child c = new Child();
    GrandChild gc = new GrandChild();
    c.m(5,true);                    c.m(6));
    gc.m(1,false);          gc.m(2));
}}
```

On the contrary, the following C++ code compiles with two errors. Both Child and GrandChild instances have only one method defined on them, and its signature is m(int, boolean). The method with signature m(int) is defined only for direct instances of Parent, and is not defined for Child and GrandChild instances.

```
class Parent {
   public:    void m(int i) {  }
};
class Child : public Parent {
   public:    void m(int i, bool b) {   }
```

```
    };
    class GrandChild : public Child {
       public:    void m(int i, bool b) {    }
    };

    int main() {
       Child* c = new Child();
       GrandChild* gc = new GrandChild();
       c->method(5,true);          c->method(6);      // ErRoR
       gc->method(1,false);        gc->method(2);     // ErRoR
    }
```

The reason for this discrepancy can be traced to the way the method calls are resolved in C++ and Java. In C++, the method name is searched starting from the class in which the method call appears, up the class hierarchy. The search stops at the first class that contains a definition of the method name m. Subsequently, C++ tries to match the entire signature of the method call m(…) using all the explicitly given overloaded definitions of m(…) in the "matched" class. If a method matching the method call signature is found, the search stops with success. Otherwise, C++ gives a compile-time error. In contrast, Java continues its search all the way to the root of the tree-structured class hierarchy to find a method that matches the method call signature. Effectively, in Java, if a method m(…) is defined in a class, it is also defined on all instances of its descendant subclasses.

Interaction of Type System with Method Inheritance in Java and C#

A method call in Java and C# is processed in two steps: The signature of the method to be called is fixed at compile-time, while the method definition to be invoked for a method call is determined at run-time. The static determination of method signature uses type coercion rules.

Consider the following Java class definitions and the driver program illustrating a variety of method calls. There is only one method in class P, while there are two methods in class C. The method f(P) defined in class P has been redefined in class C.

```
    class P {
       public void f(P p) {     }
    }
    class C extends P {
       public void f(P p) {     }
       public void f(C cp) {     }
    }

 class Calls {
 public static void
    main(String[] args) {
```

```
    P pp  = new P();         C cc  = new C();         P pc  = cc;
    pp.f(pp);  pp.f(cc);      pc.f(pp);  pc.f(cc)      cc.f(pp);  cc.f(cc);
  }
}
```

To process the call objExp.meth(arg), the compiler determines the static type of objExp, say T, and searches for a definition of meth compatible with the argument arg in the associated class T. The compiler freezes the signature of the method call at this stage. At run-time, the method code is chosen using the frozen signature in the class associated with the run-time object that objExpr evaluates to. For the call pp.f(pp), the frozen signature is f(P) and method run is the one defined in class P. For the call pp.f(cc), the frozen signature remains f(P), because the compiler searches for the definition of f(…) in class P, yielding the unique signature f(P), which is admissible as cc's type C is compatible with class P (coercion). The method run is the one defined in class P because pp refers to a P-instance. For the calls pc.f(pp) and pc.f(cc), the frozen signature remains f(P), but the method run is the one defined in class C because pc refers to a C-instance (dynamic binding). For the calls cc.f(pp) and cc.f(cc), the frozen signature is f(P) and f(C) respectively determined by searching class C, the declared type of cc, for a method signature match.

Call	Compile-time Signature	Run-time Code
pp.f(pp)	f(P) in P	f(P) in P
pp.f(cc)	f(P) in P	f(P) in P
pc.f(pp)	f(P) in P	f(P) in C
pc.f(cc)	f(P) in P	f(P) in C
cc.f(pp)	f(P) in C	f(P) in C
cc.f(cc)	f(C) in C	f(C) in C

Compilation and Interpretation of C++, Java and C#

C++ programs (*.h and *.cc files) are separately compiled into assembly language to yield object code (*.o files). These are statically or dynamically linked with library routines to obtain the executables (*.exe or a.out files) that are ready to run.

Java compiler translates source programs into machine independent Java byte code. Java 1.0 run-time for a specific platform (hardware/operating system combination) interprets Java byte code by repeatedly converting each instruction into machine code for the platform and then running it. Execution of loops could be improved by factoring out the translation of byte code into machine code and caching it first-time around the loop. This led to Java 1.1 that improves the execution efficiency by *just-in-time* compilation of the entire Java byte code program into platform-specific (native) code before running it. Subsequently, this approach was observed to cause slow start-up because of wasteful dynamic compilation of rarely used byte code segments.

The Hotspot virtual machine introduced with Java 2.0 begins as an interpreter, and through profiling determines frequently executed code segments (hotspots). Subsequently, it selectively just-in-time compiles only the "hotspots". This reduces the start-up time and improves the behavior of long-running server programs.

C# sources are compiled into Microsoft Intermediate Language (MSIL). At run-time, the MSIL code is just-in-time compiled and executed using Common Language Runtime (CLR).

In relation to object-oriented programming, it is possible to replace dynamically bound calls to the *final* methods (methods that cannot be overridden), or to methods in the terminal classes, by static binding. Hotspot virtual machine goes further in aggressively inlining small method bodies, and statically bound calls. Due to potential dynamic class loading, Hotspot virtual machine can reverse its inlining decisions if newly loaded classes extend the existing class hierarchy.

Another major aspect of modern object-oriented language run-time is the garbage collection. Refer to (Venners, 2000) for lucid details on Java virtual machine in particular and garbage collection in general.

MULTIPLE INHERITANCE

Even though the software engineering benefits of multiple inheritance are abundantly clear, the incorporation of multiple inheritance into concrete programming language is fraught with significant complexity. Essentially, undesirable problems sneak in when accommodating good examples. Thus, there is no consensus among researchers on the semantics of multiple inheritance in the presence of method overriding and potential conflicts due to multiple definitions (Meyer, 1997).

Thirunarayan et al (2001) reviews the approach taken in C++, Java, and Eiffel, and explores the patterns and the idioms used by the Java designers and programmers to redeem the advantages of multiple inheritance. The paper also discusses an alternative to multiple inheritance using constructs for type-safe automatic forwarding.

FUTURE TRENDS

The systems programming languages such as SmallTalk, Common LISP, C++, Java, C#, etc provide the necessary infrastructure for building large applications, with efficiency of execution as an important goal. With the advent of the Internet and the WWW, there has been a sudden surge of scripting languages such as TCL/TK, PERL, Python, Ruby, Visual Basic, JavaScript, PHP, etc that are primarily designed for gluing applications quickly, with programming flexibility as an important goal. Several scripting languages support advanced object-oriented programming features including multiple inheritance, prototypes and delegation, run-time addition of members to instances, meta-programming and reflection, etc. We expect

future programming languages to support features that enhance programmer productivity through code reuse, program reliability through strong typing and exception facility, rich functionality through domain-specific APIs, program efficiency through dynamic optimization, and ease of use through GUI.

C O N C L U S I O N

We selectively reviewed the basics of object-oriented programming language features and illustrated the subtleties associated with the inheritance of instance methods in widely used systems programming languages such as C++, Java and C#. For example, we explained the superiority of Java's signature-based method inheritance over C++'s use of method name-based search, C# rationale of versioning robustness for deviating from Java's instance method inheritance, and clarified Java and C#'s implementation of instance method binding using statically computed method signature and dynamically determined method definition. Overall, C++ comes across as feature-rich with lot of legacy applications, while Java and C# come across as cleaner designs for developing new applications.

In future, we expect the scripting languages popularized by improved hardware resources, and demanded by rapidly evolving, distributed and heterogeneous WWW, to support object-oriented programming features to improve programmer productivity, program reliability, functionality and ease of use.

R E F E R E N C E S

Appel, A. W. (2002) Modern Compiler Implementation in Java, Second Edition, Cambridge University Press.

Arnold, K., Gosling, J., and Holmes, D. (2005) *The Java Programming Language,* Fourth Edition, Addison-Wesley.

Budd, T. (2002) Introduction to Object-Oriented Programming, Third Edition, Addison-Wesley.

Gosling, J., Joy, B., Steele, G. and Bracha, G. (2002) The Java Language Specification. Second Edition, Addison-Wesley.

Hejlsberg, A., Wiltamuth, S., and Golde, P. (2006) *The C# Programming Language*, Second Edition, Addison Wesley.

Lakshmanan, L. V. S., and Thirunarayan, K. (1998) Declarative Frameworks for Inheritance, *Logics for Databases and Information Systems*, Eds: J. Chomicki and G. Saake, Kluwer Academic Publishers, 357-388.

Meyer, B. (1999) *Object-Oriented Software Construction*, Second Edition, Prentice Hall.

Ousterhout, J. K. (1998) Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23--30.

Stroustrup, B. (1997) *The C++ Programming Language*, Third Edition, Addison Wesley.

Wirth, N. (1988) Type extensions, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *10*(2), 204-214.

Thirunarayan, K., Kniesel, G., and Hampapuram, H. (2001) Simulating Multiple Inheritance and Generics in Java, *Computer Languages, 25*(4), 189-210.

Thirunarayan, K. (1995) Local Theories of Inheritance, *International Journal of Intelligent Systems, 10*(7), 617-645.

Venners, B. (2000) *Inside Java 2 Virtual Machine*, Second Edition, McGraw-Hill.

Terms and Definitions

Object: Object is a run-time structure with state and behavior. Object state is stored in its fields (variables) and behavior as its methods (functions).

Class: Class is a static description of objects. It defines types for fields and code for methods.

Subclass (Inheritance): Inheritance is a binary relation between classes that enables one to define a class in terms of another class 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.

Subtype (Polymorphism): A class Q is a subtype of a class P if an instance of Q can be used where an instance of P is required. A variable is said to be polymorphic if it can hold a reference to objects of different forms. Typically, a variable of type P can hold a reference to an instance of a subclass of class P.

Multiple Inheritance: A language supports multiple inheritance if a class can have multiple parents.

Static/Dynamic binding: Binding is the association of method code to a method call. Binding carried out at compile-time is called static binding, while the binding carried out at run-time is called dynamic binding.

Strong/Static/Dynamic typing: A strongly typed language guarantees that in a program all the operations are applied to operands of compatible type, and any type violation is flagged by the language implementation. A statically typed language makes such guarantees by compile-time analysis of a program. A dynamically typed language makes such guarantees using run-time checks. Modern object-oriented languages such as Java and C# are strongly typed and straddle the two extremes, by checking type constraints statically as much as possible, and generating code for performing additional type constraints at run-time in situations where those checks are data dependent.