Comp 411
Principles of Programming Languages
Lecture 17
Semantics of OO Languages

Corky Cartwright

October 26, 2012

Overview I

- In OO languages, data values (except for designated non-OO types) are special *records* [*structures*] (finite mappings from *names* to *values,* which are not necessarily objects). In OO parlance, the components of record are called *members*.
- Some *members* of an object may be functions called *methods*. Methods take **this** (the object in question) as an implicit parameter. Some OO languages like Java also support *static* methods that do not depend on **this**; these methods have no implicit parameters.
- Methods (*instance* methods in Java) can only be invoked on objects (the implicit parameter). Additional parameters are optional, depending on whether the method expects them.
- For a language to qualify as OO, it must support *inheritance*, an explicit taxonomy for classifying objects based on their members. In single inheritance, this taxonomy forms a tree; in multiple inheritance, it forms a rooted DAG (directed acyclic graph). Inheritance also provides a simple mechanism for defining some objects as extensions of others.
- Most OO languages are *class*-based (my preference because it supports a simple static type system). In *class*-based OO languages, every object is an instance of a class (an object template)

Overview II

- In *single inheritance* class-based languages, every class must declare its immediate superclass. In *multiple inheritance* class-based languages, every class must declare one or more immediate superclasses. Each superclass is either another declared class or a built-in universal (least common denominator) class [**Object** in Java]). Every class inherits all members of its immediate superclass(es); it has the option of overriding (replacing) the definitions of inherited methods.
- Java does not allow true multiple inheritance but it supports a cleaner alternative (*multiple interface inheritance*) using special classes called *interfaces.*
- The *superclass relation* is the transitive closure of the *immediate superclass relation*.
- A class cannot shadow a method defined in the parent class(es); it can only override it (replace its definition in the current class). The overriding method appears in class instances (objects) in place of the overridden one.
- A class can only shadow a field defined in the parent class; it cannot override it. Shadowing is simply the hiding of the parent field by the new fields exactly as in lexical scoping. The shadowed field still exists, but it can only be accessed by a **super** or by upcasting the type of the receiver (in a typed OO language).
- The method lookup process in OO languages is called *dynamic dispatch.* The meaning of a method call depends on the method code in **this**. In contrast, the meaning of a field reference is fixed for all subclasses of the class where the field is introduced. The field can only be shadowed but that does not affect the meaning of code that cannot see a shadowing definition.

Overview III

- Implications of overriding vs. shadowing: a *method invocation* always refers to the specified method in the receiver object even when the method has a definition in the class where the invocation appears. This mechanism is called *dynamic dispatch*; it is sometimes (misleadingly!) called *dynamic binding*.
- In contrast, *field references* refer to the field determined by lexical scoping rules (the corresponding binding occurrence of the field).
- A static type system can be used to restrict (discipline) class definitions and guarantee for all method lookups that a match will be found.
- OO languages that are *not* class-based are *prototype*-based. Any object can be a prototype (factory) for creating descendant objects.
- In prototype-based OO languages, objects literally contain methods. A lookup failure within a given object triggers a search in the ancestor object instead of the superclass.
- A prototype-based OO program can be written in a disciplined fashion (where a few factory objects function as class surrogates) so they have the same structure as a class-based program *but* type-checking is problematic. Complex static analysis is possible but it is not transparent and not very helpful (IMO) in locating and identifying program bugs.

Java Implementation I

Why talk about implementation? In real world languages, implementation dictates semantic choices. Tradeoffs!

Part I: Data representation

- Java objects include a header specifying the object class and hash code. The remaining record components [slots, fields in C parlance] are simply the members of the object. The class "identifier" is a pointer to an object (belonging to class **Class**) describing the class. How can **super** be supported? (For fields, it is trivial. For methods, use **Class** object.)
- Method lookup: simply extract the method from the receiver object. Trivial! In the absence of the method table optimization, inherited methods are simply components [slots] of the object record. But space optimization important.
- Space optimization: move (pointers to) member methods to a separate method table *for each class* which can be shared by all instances of the class. This table is part of the **Class** object for the class where the method definition appears. Note that lookup is now much more complex because only "local" methods (those explicitly defined in the object's class) are defined in the local method table. "Non-local" method references must look at superclasses.

Java Implementation II

- Interfaces can be supported in a variety of ways. Perhaps the simplest is to create a separate interface method table for each class implementing the interface. These tables are linked from the class method table. How can you find the link? Internally support getInterfaceXXX methods (dynamic dispatch)
- Observation: interface method dispatch is slower than class method dispatch.
- Fast **instanceof:** naïve implementation requires search (which can be messy if subject type is an interface). Constant time implementations are possible. One simple approach: assign consecutive indices starting at 0 to types (classes/interfaces). Maintain a bit string for each class specifying which types it belongs to. Then **instanceof** simply indexes this bitstring. Rare in well-written code; probably not worth optimizing.
- Multiple inheritance in C++ is supported by partitioning an object into sub-objects corresponding to each superclass and referring to objects using "internal" pointers so that a subclass object literally looks like the relevant superclass object. The bookkeeping (pointer adjustment) can get messy. Object pointers do not necessarily point to the base of the object! How can executing code find the base of an object (required by a cast to a different type!)? By embedding a head pointer in each sub-object representation.

Java Implementation III

Part II: Runtime

- The central (control) stack holds activation records for methods starting with **main**. There is *no* static link because Java only supports local variables.
- All objects are stored in the heap. All fields are slots in objects.
- Object values are represented by pointers (to the record representing the objects).
- Object in the heap must be reached (transitively) through local variables on the stack. In compiled code, computed values are often cached in registers.
- Instances of (dynamic) inner classes include a pointer to an enclosing parent object (static link!) so that inner class code can access fields in the enclosing object.

Java Implementation IV

- Classes are loaded dynamically by the class loader; it maps a byte stream in a file to a class object including code for all of the methods. The class loader performs *byte code verification* to ensure the loaded classes are well-formed and type-correct. In Java systems using "exact" GC, the class loader must build stack maps (indicating which words in the current activation record are pointers) for a sufficient set of program "safe points" also called "consistent regions". There is not a single stack map for each method because local variable usage can vary during method execution! (Allowing this variance was a bad design decision in my opinion!) Class loading of referenced classes (except superclasses) can be lazy.
- The Java libraries are simply an archive (typically in zip file format) containing a file tree of class files (byte streams).
- Java allows programs to use custom class loaders. My research group's NextGen compiler supporting first-class generics critically depends on this fact. So does DrJava (for different reasons).
- Header optimization: use the pointer to the method table as the class "tag"; method table must contain pointer to the **Class** object. Method table also includes a pointer to the superclass method table.

Java Criticisms

- Not truly OO:
  - Values of primitive types are not objects (machine level data representation should be an optimization)
  - Static fields and methods (useful in practice just like mutation functional languages)
- Interfaces are not fully satisfactory as replacement for multiple inheritance.

  Interfaces should be generalized to "traits" which are classes with no fields. The complexity of multiple inheritance is due to the fact that the same field can be inherited in multiple ways (the "diamond" relationship). This pathology cannot occur in multiple trait inheritance.
- Type system is too baroque and too restrictive.

  Generic (parameterized) nominal type systems are still not fully understood. When Java was invented, nominal OO typing was still a radical idea. (It was present in Eiffel, C++, and Objective C which were not type safe! Eiffel subsequently added an ugly runtime check to salvage type safety).
- Excessive generalization of some constructs and mechanisms leads to baroque language specification.
  Examples: 1. **<receiver>.new <type>(...)**
  2. **new <InnerClassType>(...)** outside of enclosing class
  3. unrestricted wildcard types (wildcards as bounds!)
- Erasure based implementation of generic types.
  A huge (!) mistake IMO but note that I am clearly biased.
- The run-time check in array element updating is awkward.

  The designers wanted co-variant subtyping for arrays (u <= v implies u[] <= v[]) which is important in absence of generic types. Operations that take arbitrary arrays of reference type as arguments can be written in terms of Object[]--assuming that such arguments are not mutated. (Mutating an array of type T[] that is bound to a parameter of type Object[] can generate a run-time error. Co-variant subtyping is difficult to support in program defined classes because method input types behave contra-variantly in subtyping relationships!). Java 5/6/7 uses wildcard types to support co-variance but did not get the details right. C# does not support any covariance.

Directions for Further Study

- Custom class loaders. Do a web search on Java class loading. The articles at onjava.com are particularly good.
- Nominal Type Systems. I emphatically disagree with the comments

Swarat made about the differences between inheritance and subtyping. His comments represent the mainstream view of PL researchers but I believe this view is wrong (misleading) because it presumes a structural model of OOP and type system corresponding to such a structural model. Structural models of OOP are extensions of models of FP where objects are simply records. The name of a class (and its entire signature) has nothing to do with its meaning. You cannot explain nominal subtyping (as in Java) using such a model. Such models are bogus and cannot be extended to accurate models of Java programs. (I may give a pizza talk on this subject in the near future.)

- Other languages that generate code for the JVM: Scala, Groovy, Kotlin, …
- The essence of the Java platform is the JVM. It will outlive me and perhaps you. I hope that we see a few minor but critical extensions.