Comp 411
Principles of Programming Languages
Lecture 19
Semantics of OO Languages

Corky Cartwright

Mar 7-11, 2022

Overview I

- In OO languages, OO data values (except for designated *value* [non-OO] types), are special *records* [*structures* in Scheme] (finite mappings from *names* to *values*). In OO parlance, the components of record are called *members*. In C/C++, they are called *fields*, which is confusing because *field* has a different meaning in OO languages.
- Some *members* of an object may be functions/procedures called *methods*. Methods are procedures 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. Static methods literally are procedures! In efficient OO language implementations, method members are shared since they are the same for all instances of a class, but this sharing is an optimization in statically typed OO languages since the collection of methods in a class is immutable during program evaluation (computation). The object members that are not methods are called *fields*. They are (typically *mutable*) cells.
- A true method (*instance* method in Java) can only be invoked on an object (the *receiver*, an implicit parameter). Additional parameters are optional, depending on whether the method expects them. This invocation process is called *dynamic dispatch* because the executed code is literally extracted from the object: *the code invoked at a call site depends on the value of the receiver*, which can change with each execution of the call.

Overview II

- A language with objects is OO if it supports *dynamic dispatch* (discussed in more detail in Overview II & III) and *inheritance*, an explicit taxonomy for classifying objects based on their members and class names where superclass/parent methods are inherited unless overridden.
- In single inheritance, this taxonomy forms a tree. In multiple inheritance, it forms a rooted DAG (directed acyclic graph) where the root class is the universal class (**`Object`** in Java).
- 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) and includes a tag field identifying the class to which the object belongs. The class of an object completely determines its structure, namely the members of the object and their types (which we will discuss in depth in a few weeks).
- Other OO languages are *prototype*-based where objects are *cloned* (copied) to create new objects; the bindings of method names can be *updated* (which is disallowed in nearly all class-based OO languages) and members can be *dynamically added* to objects during program execution (also disallowed in nearly all class-based OO languages.) Inheritance occurs because of cloning. These two mechanisms, *object cloning* and *method adding* make the static typing of such languages problematic; nearly all prototype languages (like Javascript) are dynamically typed.

Overview III

- In *single inheritance* class-based languages, every class must declare a unique 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 definition inherits all members of its immediate superclass(es); it also 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* (which in Java 8+ can contain concrete methods; such generalized interfaces are often called *traits*).
- 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 via `super` (an ugly variant of `this`) 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 IV

- Implications of overriding vs. shadowing: a *method invocation* always refers to the specified method in the receiver object (`this`) 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*. Dynamic dispatch combined with inheritance to share common code among variants is the essence of OO.
- In contrast, *field references* refer to the field determined by lexical scoping rules (the corresponding binding occurrence of the field). Hence, direct field references should only be local in well-written OO code. In addition, fields introduced in subclasses should never match the field names of superclasses.
- A static type system can be used to restrict (discipline) class definitions and guarantee for all method lookups (assuming the receiver is not `null`) that a match will be found. Even the issue of `NullPointerException`s can be addressed by supporting types that exclude `null`.
- OO languages that are *not* class-based are *prototype*-based. Any object can be a prototype (factory) for creating cloned objects. In such languages, inheritance is pervasive but it is hidden in the code as prototype cloning and augmentation.
- In prototype-based OO languages, objects literally contain methods. A lookup failure within a given object triggers a search in the ancestor object (creator) instead of the superclass. This is very ugly from the perspective of program design and program implementation. I have never seen a programming logic for a prototype-based OO language. Prototype-based OO languages make program verification nearly impossible because the validity of method dispatch depends on the execution history of the program (which dynamically builds the inheritance hierarchy). Ugh!
- A prototype-based OO program can be written in a disciplined fashion (where a few factory objects function as class surrogates) so they have essentially the same structure as a class-based program *but* type-checking is still problematic and the clarity of the code is highly dependent on the author.
- Complex static analysis is possible but it is not transparent and not very helpful (IMO) in locating and identifying program bugs. The experimental research language Cecil did this.

Overview V

Thumbnail History of the Evolution of OO Languages:

Simula (1967)

- Allows entire Algol blocks to be autonomous data values with independent lifetimes foreshadowing objects.
- Classes can be formulated as special procedures that return blocks.
- Allows autonomous blocks to be defined as extensions of other blocks; inheritance = lexical scoping + copying! Inheritance is single because it is block extension.
- No conventional overriding but inner calling mechanism (similar to **`super`** in Java) resembles the inverse of overriding.
- Incorporates some important software engineering insights but there is no clear design methodology underlying the language, nor is there a credible programming logic.

Smalltalk (1972)

- Dynamically typed
- Supports reflective access to the runtime (essentially the same mechanism as reflection in Java and other newer languages).
- Single inheritance.
- Dynamic extension of objects.
- If dynamic features are exploited, software engineering is compromised. Clean language after patching the semantics of “blocks” (closures) provided dyanamic changes to objects is shunned.

Self (1987)

- Dynamically typed
- Prototype based
- Activation records are objects (the same identification as in Simula but it is posited in reverse).
- Dynamic scoping except for explicit closures (Ugh!).
- Introduced generational GC.

Overview VI

Impact on Contemporary Languages and Software Technology

- Static typing and dynamic scoping is a toxic combination. Challenge: devise a statically typed dynamically scoped language. Essentially impossible without emasculating dynamic scope.
- Pragmatic OO extensions of C: C++ and Objective C. Truly OO except storage management is manual, which is a crippling software engineering defect IMO. Nominal, unsound type system is principal innovation. (Right idea; imperfect execution.) Subsequently made sound in Java and Swift.
- Important distinction: structural subtyping (ML and other pedagogic extensions) vs. nominal (C++, Java, Scala, Swift, etc.)
- Pedagogic OO Extensions of ML culminating in OCaml; not truly OO because type system interferes with OO design. The OCaml type system tries to formulate inheritance-based subtyping as polymorphism, which often breaks in practice.
- OO features of ML with objects (Ocaml) are usually ignored by MLers ; structural typing and OO do not mix well.
- Eiffel (statically-typed forerunner of Java)/Dylan (Scheme with classes and inheritance)
- Modern OO descendants: Java/C#/Scala/Swift
- Important distinction: structural subtyping (ML and extensions) vs. nominal (C++, Java, Scala, Swift, etc.). Semantics of nominal types is surprisingly different than semantics of structural types.
- OO features of ML with objects (Ocaml) are usually ignored by MLers ; structural typing and OO do not mix well

Java as a Real OO language

- Java is most important practical OO language.
- Two views
  - C++ augmented by GC
  - Dylan (Scheme + objects) with a C-like syntax and static type checking. Scheme (Racket) now has a gradual static type system.
- I strongly support the latter view. Why? The semantic definitions of C++ and Java are completely different while the semantics of Dylan and Java are very similar. It is easy to transliterate Scheme (sans `call/cc`) into Java. It is essentially impossible to transliterate C++ into Java (unless the C++ code is written with this form of translation in mind).

Java Implementation I

Why talk about implementation? In real world languages, implementation impacts program design. There are performance and design tradeoffs!

Part I: Data representation

- Java objects include a header specifying the object class and hash code (lazily generated on demand using the object address). The remaining record components [slots, fields in C parlance] are simply the members of the object. The "identifying code" for a class 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 the `Class` object.)
- Method lookup: simply extract the method from the receiver object using its known offset within the `Class` object. Method offsets are consistent across subclasses because all new members added in subclasses are added at the "end" of the inherited object format. In the absence of the method table optimization, inherited methods are simply components [slots] of the object record. But space optimization is important. Full multiple inheritance (as in C++) requires some magic (embedded objects) to support this implementation.
- 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 the method table can contain an entry for every method of the class including inherited methods. If inherited methods are not included, method 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.

Java Implementation II

- Linking (resolving symbolic references) is done dynamically during execution! In most mainstream languages supporting separate compilation (like C++), it is done statically by a linker prior to program execution.
- 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 hidden **`getInterfaceXXX`** methods (dynamic dispatch) in a class method table for all interfaces implemented by the class.
- Observation: interface method dispatch is slightly slower than class method dispatch because it requires two dynamic dispatches.
- 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 in the **`Class`** object.
- 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 because object pointers do not necessarily point to the base of the object! How can the executing code find the base of an object (required by a cast to a different type!)? By embedding a hidden head pointer in each sub-object representation. GC marking obviously becomes more complex.

Java Implementation III

Part II: Runtime

- The central (control) stack holds activation records for methods starting with the `main` method for the root class. There is *no* static link because Java only supports local variables. Lexical nesting only occurs via inner class declarations; a link to the enclosing object is embedded in each instance of an inner class. These links form a static chain!
- All objects are stored in the heap. All fields are slots in heap objects.
- Object values are represented by pointers (to records representing the objects).
- Objects in the heap can only be reached (transitively) through local variables on the stack, static fields, object fields in the heap and computed pointers (such as the result returned by a `new` operation). In compiled code, computed values are often cached in registers. "Live heap memory" is the set of objects at locations generated by the transitive closure of the "refers-to heap location relation" starting with the "root" references in the stack, registers, and static memory areas.
- 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!) Newer JVMs embed stack maps in class files for faster class loading. Suggested revision to the JVM: do not share activation record slots across disjoint variables so the stack map information for an object is static! Actually, only need to prevent sharing between variables of value type and reference type.
- 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. Our NextGen compiler which supported first-class generics critically depended on this fact. So does DrJava (for different reasons). Custom class loaders can potentially support macros.
- Header optimization: use the pointer to the method table as the class "identifier"; method table must contain pointer to the `Class` object.
- The method table also includes a pointer to the superclass method table.

Java Criticisms

- Not fully OO:
  - Values of primitive types are not objects (should be hidden)
  - Static fields and methods (useful in practice just like mutation in functional languages) but ugly. Scala handles this issue much better by introducing a singleton object for every class (distinct from the conventional hidden class object).
- Interfaces were generalized to "traits" which may contain concrete methods bu the syntax for such methods is ugly (with the misleading modifier **`default`**) and discourages proper usage. The "trait" idea was pioneered in a rewriting of the SmallTalk libraries. 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 both baroque and too restrictive. Generic (parameterized) nominal type systems are still not fully understood. When Java was invented, nominal typing was still a radical idea, used in Eiffel, C++, and Objective C which were not type safe. Eiffel subsequently added an ugly runtime check to technically salvage type safety. Scala sets the standard in this arena.

Java Criticisms cont.

- Excessive generalization of some constructs and mechanisms leads to a baroque language specification (which creates lots of buzz for language "lawyers" who know the details of the language specification but perhaps nothing about good OO design). Some examples:
  - ***`receiver.new type(...)`*** when ***`type`*** is shadowed or has the receiver type as the enclosing class
  - ***`newInnerClassType(...)`*** outside of the enclosing class (which require an extra argument!)
  - excessively general local type inference for polymorphic methods
  - unrestricted wildcard types (wildcards as bounds!)
- Erasure-based implementation of generic types. A huge (!) mistake IMO.
- The run-time type check in array element updating is awkward.
- The designers wanted co-variant subtyping for arrays (**`u`** <= **`v`** implies **`u[]`** <= **`v[]`**) which is important in the absence of generic types. Covariant subtyping is difficult to support in an OO language (because method input types behave contravariantly in subtyping relationships!). Java 5+ uses wildcard types to support co-variance but did not get the details right. C# initially shunned co-variance but eventually followed Scala's lead and adopted explicit co- and contra-variance in class declarations. Scala is the standard IMO. Swift generics are a mess.
- The reliance on a virtual machine and JIT compilation makes Java applications unnecessarily heavy (slow to start and not particularly fast in common usage). In contrast, Swift is a mess; I don't think the designers understand the concepts of co-variance and contra-variance of type parameters; nor do they understand upper and lower parameter bounds.

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 prevailing view among PL theorists (almost none of whom I am convinced understands real world OO design) who claim *inheritance is not subtyping*. It is easy to erroneously define subclasses that break inherited contracts. But it does not justify elevating this design error to a feature! (Sound familiar?). If you hear the terms "binary methods" (methods in a class `C` with an argument of type `C`) or "implementation inheritance" (the sanitized name used for classes that break contracts), you are listening to a language theorist who does not understand how to reason about OO programs or recognize that it is important. They are typically condescending functional programming advocates. (I used to be one of them until I had to learn Java to teach it. The view that "inheritance is not subtyping" is now part of the folklore of PL research. I have no quarrel with this mantra if you add the clause "for a PL research hack who knows nothing about OO design."
- Other languages that generate code for the JVM: Scala, Groovy, Kotlin, ...
- The essence of the Java platform is the JVM. It will probably outlive me and perhaps you. It can certainly be improved but I am growing disenchanted with the JVM. Why? It has such a long startup time, defers important program optimization to runtime, and wastes so much energy.
- Can we support a programming ecosystem as rich as what Java provides without a virtual machine. Swift may be such a framework but the departure of Chris Lattner from Apple has (IMO) crippled the evolution of the language.