Overview I

- In OO languages, OO data values (except for designated non-OO types) are special *records* [structures] (finite mappings from *names* to *values*). 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.

- A method (*instance* method in Java) can only be invoked on an object (the implicit parameter). Additional parameters are optional, depending on whether the method expects them.

- A language with objects is OO if it supports *inheritance*, an explicit taxonomy for classifying objects based on their members and class names.
  - 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.) These two mechanisms make the static typing of such languages problematic; nearly all (like Javascript) are dynamically typed.
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 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 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 cloned objects.

- 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. I have never seen a programming logic for a prototype-based OO language. Prototype-based OO languages make program verification nearly impossible because method dispatch depends on the execution history of the program. 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 the same structure as a class-based program but type-checking is still problematic.

- Complex static analysis is possible but it is not transparent and not very helpful (IMO) in locating and identifying program bugs.
Overview IV

Thumbnail History of the Evolution of OO Languages:

Simula (1967)
- Allows 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 mechanism resembles the inverse of overriding
- Incorporates some important software engineering insights but no there is clear design methodology underlying the language

Smalltalk (1972)
- Dynamically typed
- Class based
- 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

Self (1987)
- Dynamically typed
- Prototype based
- Activation records are objects (Simula in reverse!)
- Dynamic scoping except for explicit closures (Ugh!)

Aside; devise a statically typed dynamically scoped language. Essentially impossible without emasculating dynamic scope

Pedagogic OO Extensions of ML culminating in OCaml; not truly OO because type system interferes with OO design.

Pragmatic OO extensions of C: Objective C, C++; truly OO except storage management is manual, which is a headache

Eiffel/Dylan (Scheme with classes and inheritance)/Java/C#/Scala/Swift

Important distinction: structural subtyping (ML and other pedagogic extensions) vs. nominal (C++, Java, Scala, Swift, etc.)
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 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 program with this form of translation in mind).
Java Implementation I

Why talk about implementation? In real world languages, implementation dictates semantic choices. There are performance and design 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 search up the the superclass chain (which has no connection with the static and dynamic chains that may exist the run-time stack [assuming stack-managed activation records].

• 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 slightly 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.

• 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.
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 heap objects.

Object values are represented by pointers (to records representing the objects).

Objects in the heap must be reached (transitively) through local variables on the stack, 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.

- 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 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 “identifier”; method table must contain pointer to the `Class` object.

- The method table also includes a pointer to the superclass method table.
Java Implementation V

• In the presence of the method table optimization (which is essentially universal), objects only contain header information and fields (in OO parlance).

• 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 using dynamic dispatch (essentially a hidden `getInterfaceXXXTable()` method).

• 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) and maintain a bit string for each class specifying which types it belongs to. Then `instanceof` simply indexes this bitstring. Must be updated when a new class is loaded (if class indices monotonically increase, can use demand-driven updates based on index out-of-bounds exceptions).
Java Criticisms

- Not truly OO:
  - Values of primitive types are not objects (should be hidden)
  - Static fields and methods (useful in practice just like mutation in functional languages)
- Interfaces are not fully satisfactory as replacement for multiple inheritance. Interfaces should be generalized to “traits” which are classes with no fields (which was finally done in Java 8 but the syntax is awkward). This 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 (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 a baroque language specification (which creates lots of buzz for language “lawyers” who know the details of the language specification but 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 enclosing class
  - excessively general local type inference for polymorphic methods
  - unrestricted wildcard types (wildcards as bounds!)
- Erasure based implementation of generic types.
  - A huge (!) mistake IMO but note that I am 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 the absence of generic types. Co-variant subtyping is difficult to support in an OO language (because method input types behave contra-variantly in subtyping relationships!). Java 5+ uses wildcard types to support co-variation 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 prevailing view among PL theorists (none of whom I am convinced understands real world OO design) that inheritance is not subtyping. It is easy to 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 that 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 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 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.