Support for Object-Oriented Programming

COS 301 — Programming Languages

- Chapter 12
- Slides draw heavily on Sebesta's slides

OOP

- Object-oriented programming, according to the person who invented the term (Alan Kay), needs: (from http://community.schemewiki.org/?object-oriented-programming)
  - **Actors model** — basically, "actors" (objects) respond to messages as they locally see fit; not a function call situation
  - Encapsulation
  - Protection
  - Ad hoc polymorphism
  - NO inheritance

OOP

- According to Kay,

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

- Maybe a bit extreme for modern tastes

OOP

- Objects — special kind of abstract data type
- Encapsulate both data and process
- Some OOP languages support imperative programming: e.g., Ada, C++, Swift
- Some support functional programming — e.g., Lisp/CLOS, Lisp/Flavors, Scheme's various object-system add-ons, Racket
- Some languages don't support other paradigms, but use imperative structures — e.g., Java, C#
- Some are pure OOP — e.g., Smalltalk, Ruby

Object-oriented programming

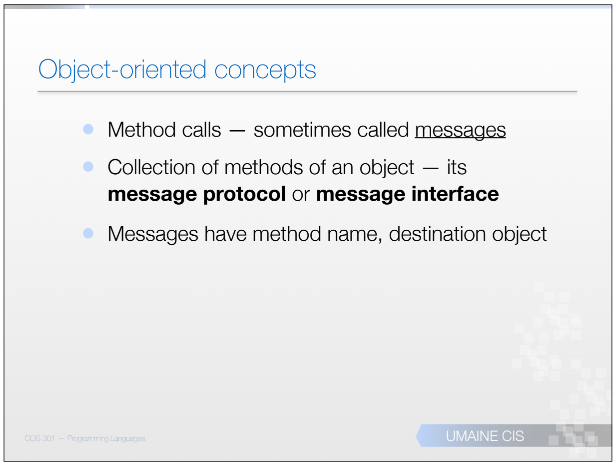
- Three major language features:
  - Abstract data types
  - Inheritance — central theme in OOP and OOP languages (contra Kay)
  - Polymorphism

Inheritance

- **Inheritance** — new classes defined in terms of existing ones → inherit common parts
- Allows reuse of ADTs with changes — may be difficult without, since ADTs often need changes to be made to work for particular application
- Defines classes in a hierarchy — ADTs are all independent and at same level
- Reuse ⟹ productivity increases

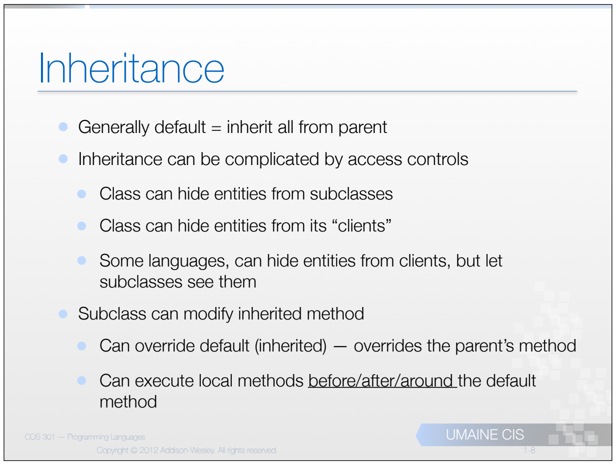
Object-oriented concepts

- ADTs are usually called classes
- Class instances are called objects
- Subclass or *derived class* — inherits from parent (superclass)
- Subprograms that operate on (belong to) objects = methods
- Variables encapsulated by objects = instance variables



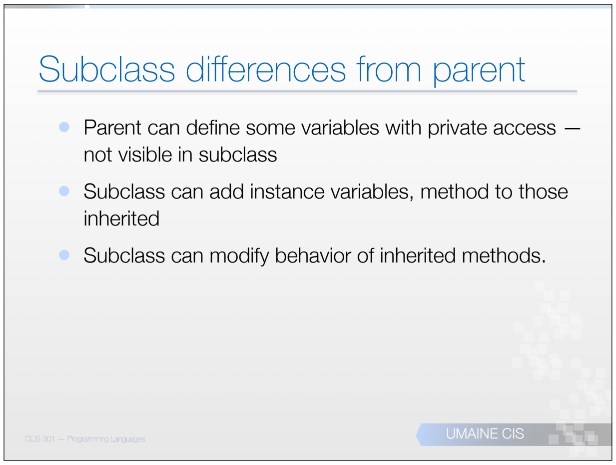
Object-oriented concepts

- Method calls — sometimes called messages
- Collection of methods of an object — its **message protocol** or **message interface**
- Messages have method name, destination object



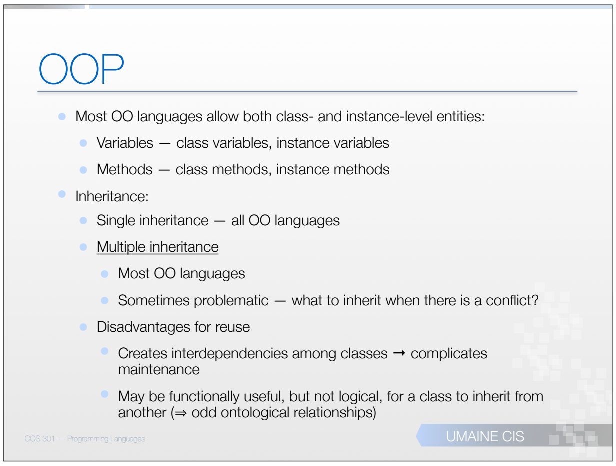
Inheritance

- Generally default = inherit all from parent
- Inheritance can be complicated by access controls
  - Class can hide entities from subclasses
  - Class can hide entities from its “clients”
  - Some languages, can hide entities from clients, but let subclasses see them
- Subclass can modify inherited method
  - Can override default (inherited) — overrides the parent’s method
  - Can execute local methods before/after/around the default method



Subclass differences from parent

- Parent can define some variables with private access — not visible in subclass
- Subclass can add instance variables, method to those inherited
- Subclass can modify behavior of inherited methods.



OOP

- Most OO languages allow both class- and instance-level entities:
  - Variables — class variables, instance variables
  - Methods — class methods, instance methods
- Inheritance:
  - Single inheritance — all OO languages
  - Multiple inheritance
    - Most OO languages
    - Sometimes problematic — what to inherit when there is a conflict?
  - Disadvantages for reuse
    - Creates interdependencies among classes → complicates maintenance
    - May be functionally useful, but not logical, for a class to inherit from another (⇒ odd ontological relationships)

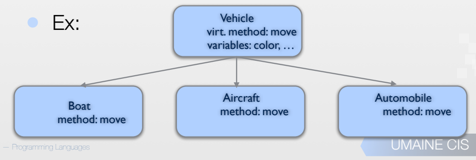
Dynamic binding in OOP

- Since a hierarchy of classes exist, can exploit for polymorphism
- **Polymorphic variable:** can hold objects of a class or any of its descendants
- Can even point to top of object hierarchy → <u>any</u> object
- **Dynamic binding:**
  - Some methods of some subclasses may override a parent's
  - Which method of which class is called is decided at run-time
- Benefits:
  - The usual ones for polymorphism
  - Also: easy to extend software system during development and maintenance

---

Dynamic binding

- **Abstract (virtual) method:** only defines a protocol, not a definition
- **Abstract class:**
  - Includes at least one abstract method
  - Cannot be instantiated
- Ex:



Vehicle
virt. method: move
variables: color, …

Boat
method: move

Aircraft
method: move

Automobile
method: move

OOP design issues

- Is everything an object?
- Subclasses = subtypes?
- Single or multiple inheritance?
- Allocating and deallocating objects
- Dynamic and static binding
- Nested classes?
- Object initialization

Exclusivity of objects

- Some languages: <u>everything</u> is an object — e.g., Ruby, Smalltalk
  - Advantage: elegance, purity, homogeneity of all data structures
  - Disadvantage: can be slow for simple objects
- Other languages: objects are added to a complete typing system — Lisp, Python,…
  - Advantage - fast operations on simple objects
  - Disadvantage - results in a confusing type system (two kinds of entities)
- Other languages: use imperative-style typing system for primitives, but everything else is an object — Java, Swift, …
  - Advantage - fast operations on simple objects and a relatively small typing system
  - Disadvantage - still some confusion because of the two type systems

Are (sub)classes (sub)types?

- Most OO languages: yes
- Basically: does an “**isa**” relationship hold between parent class and subclass?
- If so, then instances of subclass must behave the same (more or less) as instances of the parent
- Subclass can only:
  - Add variables and methods
  - Override methods in “compatible” ways
- Also has some implication for **ontology** the programmer has in mind
  - Subclasses are made for ontological reasons, not just for functionality and reuse
  - E.g., make airplane subclass of vehicle, not bird — even though “fly” method could be inherited in the latter

---

Single or multiple inheritance?

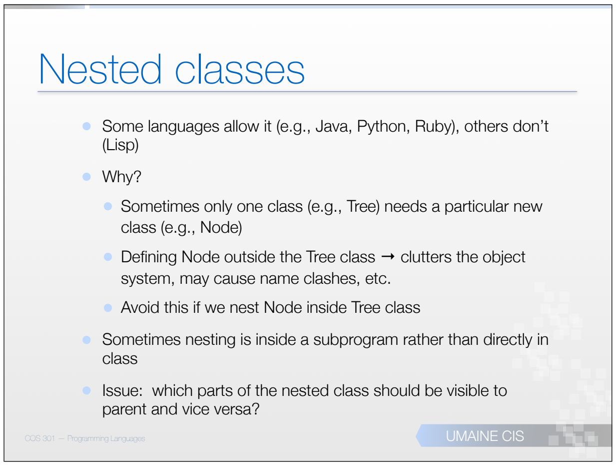
- Advantages of multiple inheritance:
  - convenient — methods, variables from multiple sources
  - ontologically-useful — aircraft isa vehicle, flying-object, bird isa animal, flying-object
- Disadvantages:
  - Complexity of language, implementation (e.g., handling name collisions)
  - Potential inefficiency — increased cost of dynamic binding (search problem)

Object allocation & deallocation

- Where do objects live?
  - If treated as other ADTs, can be allocated anywhere: run-time stack, heap (via `new`, e.g.)
  - If heap-dynamic only (e.g., Java, Lisp, Python,…)
    - References can be uniform via pointer/reference variable
    - Simplifies assignment; dereferencing can be implicit
  - If stack-dynamic only: can ⇒ object slicing
    - Object of subclass A may be larger than one of its parent class B
    - Suppose subroutine expects instance of B…
    - …pass instance of A…
    - …not enough room allocated, some instance variables not copied — or worse
    - Kind of unavoidable with call-by-value and polymorphism by classes
- Deallocation: automatic (GC) or explicit?

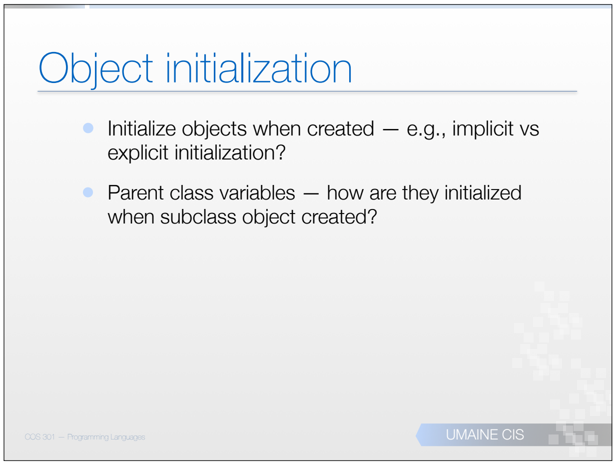
Dynamic and static binding

- Static binding — can’t do polymorphism using classes
- Dynamic binding — can be inefficient
- Maybe: allow user to specify



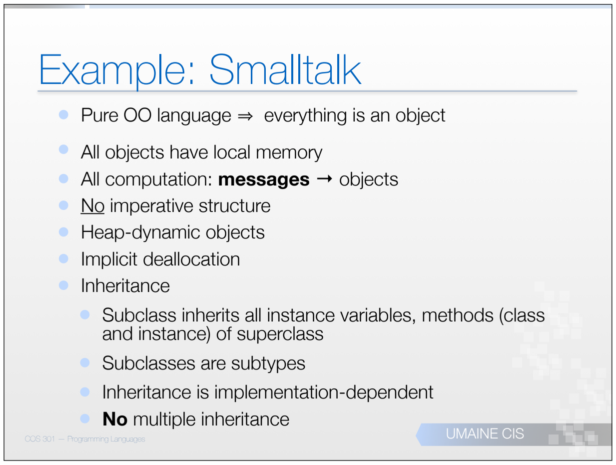
Nested classes

- Some languages allow it (e.g., Java, Python, Ruby), others don't (Lisp)
- Why?
  - Sometimes only one class (e.g., Tree) needs a particular new class (e.g., Node)
  - Defining Node outside the Tree class → clutters the object system, may cause name clashes, etc.
  - Avoid this if we nest Node inside Tree class
- Sometimes nesting is inside a subprogram rather than directly in class
- Issue: which parts of the nested class should be visible to parent and vice versa?



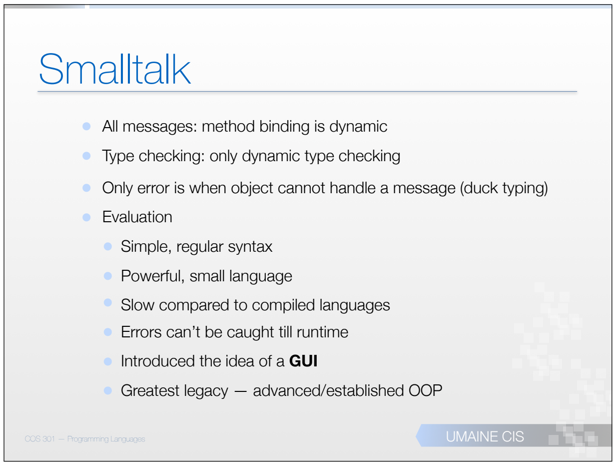
Object initialization

- Initialize objects when created — e.g., implicit vs explicit initialization?
- Parent class variables — how are they initialized when subclass object created?



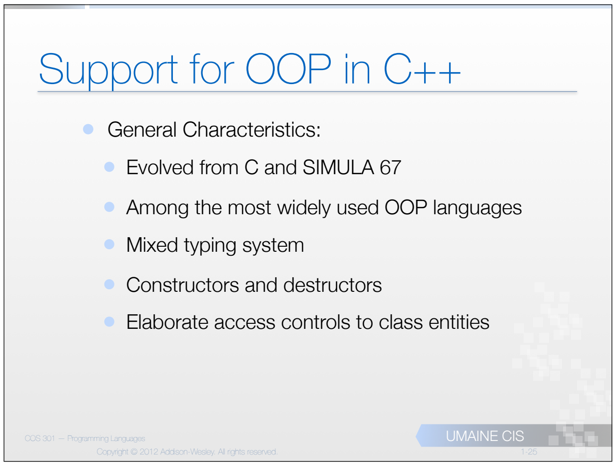
Example: Smalltalk

- Pure OO language ⇒ everything is an object
- All objects have local memory
- All computation: **messages** → objects
- No imperative structure
- Heap-dynamic objects
- Implicit deallocation
- Inheritance
  - Subclass inherits all instance variables, methods (class and instance) of superclass
  - Subclasses are subtypes
  - Inheritance is implementation-dependent
  - **No** multiple inheritance



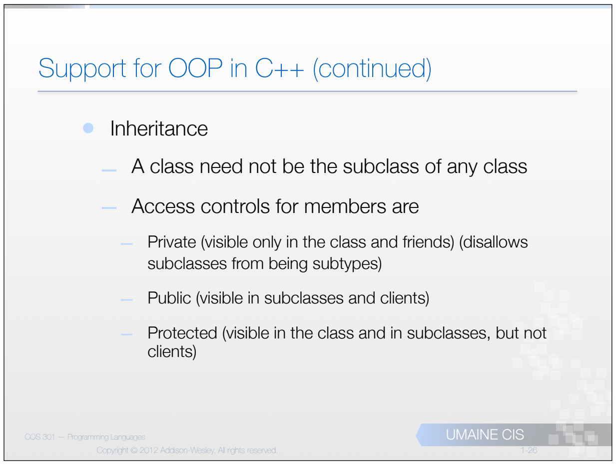
Smalltalk

- All messages: method binding is dynamic
- Type checking: only dynamic type checking
- Only error is when object cannot handle a message (duck typing)
- Evaluation
  - Simple, regular syntax
  - Powerful, small language
  - Slow compared to compiled languages
  - Errors can't be caught till runtime
  - Introduced the idea of a **GUI**
  - Greatest legacy — advanced/established OOP



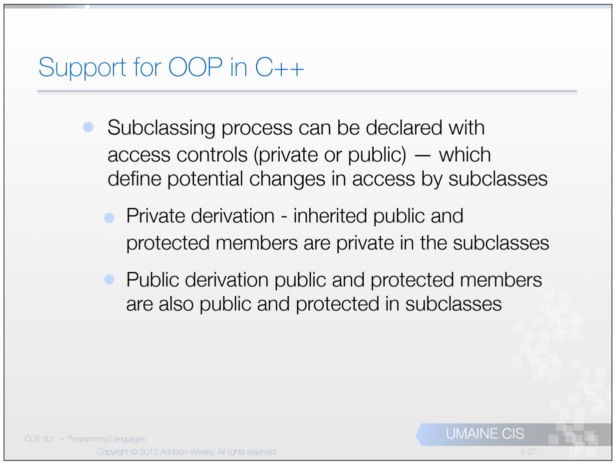
Support for OOP in C++

- General Characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities



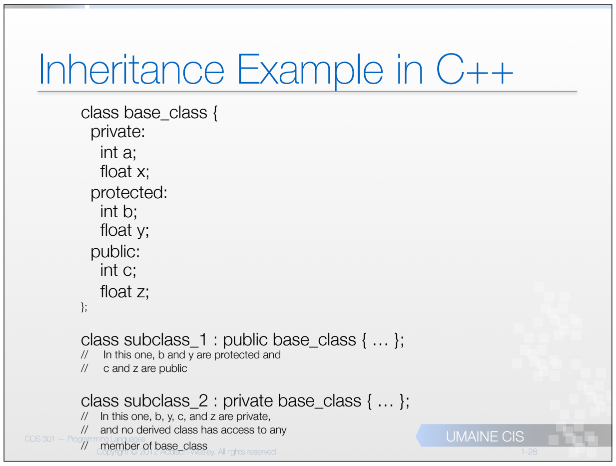
Support for OOP in C++ (continued)

- Inheritance
  - A class need not be the subclass of any class
  - Access controls for members are
    - Private (visible only in the class and friends) (disallows subclasses from being subtypes)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)



Support for OOP in C++

- Subclassing process can be declared with access controls (private or public) — which define potential changes in access by subclasses
  - Private derivation - inherited public and protected members are private in the subclasses
  - Public derivation public and protected members are also public and protected in subclasses



Inheritance Example in C++

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class { … };
//    In this one, b and y are protected and
//    c and z are public

class subclass_2 : private base_class { … };
//   In this one, b, y, c, and z are private,
//   and no derived class has access to any
//   member of base_class
```

Re-exportation in C++

- Member not accessible in a subclass (because of private derivation) → can be declared to be visible there using the scope resolution operator (::), e.g.,

```
class subclass_3 : private base_class {
     base_class :: c;
          …
}
```

Re-exportation

- One motivation for using private derivation
  - Class provides members that must be visible → public members
  - Derived class adds some new members, but does not want its clients to see parent's members

---

Support for OOP in C++ (continued)

- Multiple inheritance
  - Two inherited members with the same name: both can be referenced using the scope resolution operator (::)

```
class Thread { ... }

class Drawing { ... }

class DrawThread : public Thread, public Drawing {
  … }
```

Support for OOP in C++ (continued)

- Dynamic Binding
  - *Virtual method:* can be called through polymorphic variables and dynamically bound to messages
  - Pure virtual function has no definition at all
  - Class that has at least one pure virtual function is an abstract class

Support for OOP in C++ (continued)

- Evaluation
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance
  - In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound
    - Static binding is faster!
  - Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  - Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

Support for OOP in Objective-C

- Like C++, Objective-C adds support for OOP to C
- Design was at about the same time as that of C++
- Largest syntactic difference: method calls are **messages**
- Interface section of a class declares the instance variables and the methods
- Implementation section of a class defines the methods
- Classes cannot be nested

Support for OOP in Objective-C

- Inheritance
  - Single inheritance only
  - Every class must have a parent
  - NSObject is the base class

```
@interface myNewClass: NSObject { … }
…
@end
```

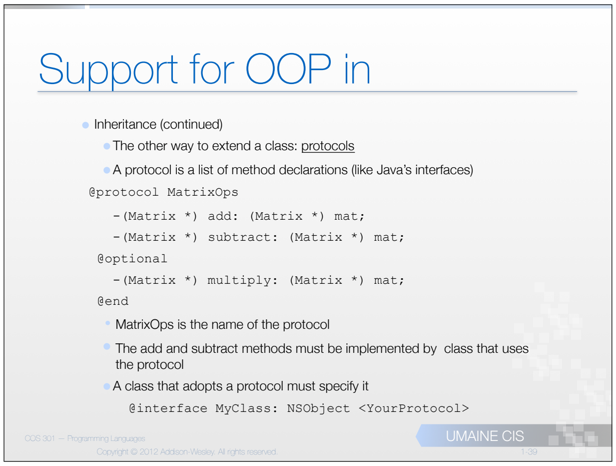
  - Because base class data members can be declared to be private, subclasses are not necessarily subtypes
  - Any method that has the same name, same return type, and same number and types of parameters as an inherited method overrides the inherited method
  - An overriden method can be called through `super`
  - All inheritance is public (unlike C++)

Support for OOP in Objective-C

- Inheritance (continued)
- Objective-C has two approaches besides subclassing to extend a class
  - A category is a secondary interface of a class that contains declarations of methods (no instance variables)

```
#import ″Stack.h″
@interface Stack (StackExtend)
  -(int) secondFromTop;
  -(void) full;
@end
```

  - A category is a mixin – its methods are added to the parent class
  - The implementation of a category is in a separate implementation: @implementation Stack (StackExtend)



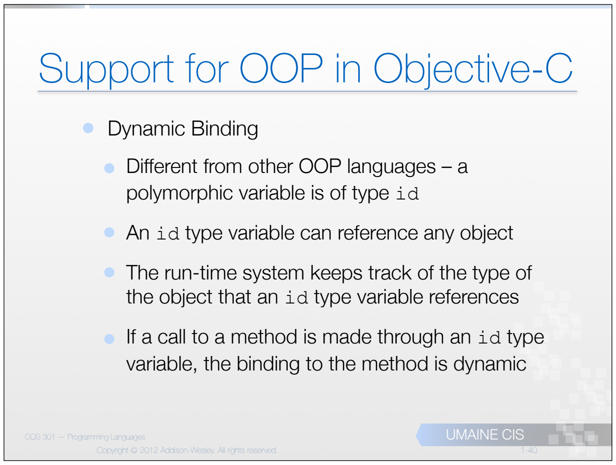
Support for OOP in

- Inheritance (continued)
  - The other way to extend a class: protocols
  - A protocol is a list of method declarations (like Java's interfaces)

```
@protocol MatrixOps
   -(Matrix *) add: (Matrix *) mat;
   -(Matrix *) subtract: (Matrix *) mat;
 @optional
   -(Matrix *) multiply: (Matrix *) mat;
 @end
```

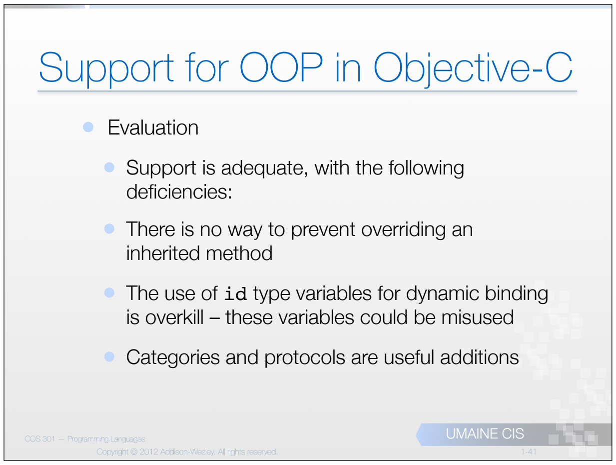
  - MatrixOps is the name of the protocol
  - The add and subtract methods must be implemented by class that uses the protocol
  - A class that adopts a protocol must specify it

```
@interface MyClass: NSObject <YourProtocol>
```



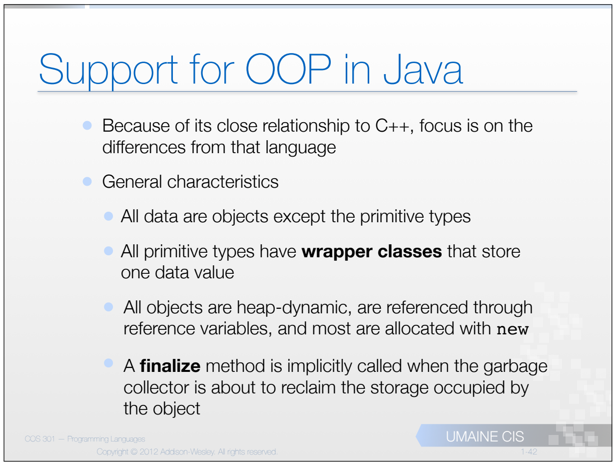
Support for OOP in Objective-C

- Dynamic Binding
  - Different from other OOP languages – a polymorphic variable is of type `id`
  - An `id` type variable can reference any object
  - The run-time system keeps track of the type of the object that an `id` type variable references
  - If a call to a method is made through an `id` type variable, the binding to the method is dynamic



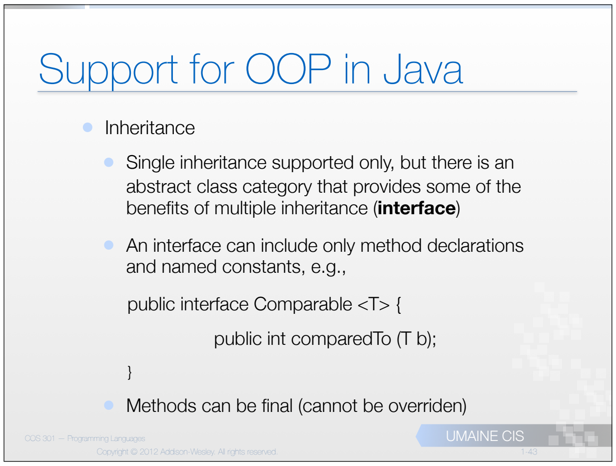
Support for OOP in Objective-C

- Evaluation
  - Support is adequate, with the following deficiencies:
  - There is no way to prevent overriding an inherited method
  - The use of **`id`** type variables for dynamic binding is overkill – these variables could be misused
  - Categories and protocols are useful additions



Support for OOP in Java

- Because of its close relationship to C++, focus is on the differences from that language
- General characteristics
  - All data are objects except the primitive types
  - All primitive types have **wrapper classes** that store one data value
  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with **`new`**
  - A **finalize** method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object

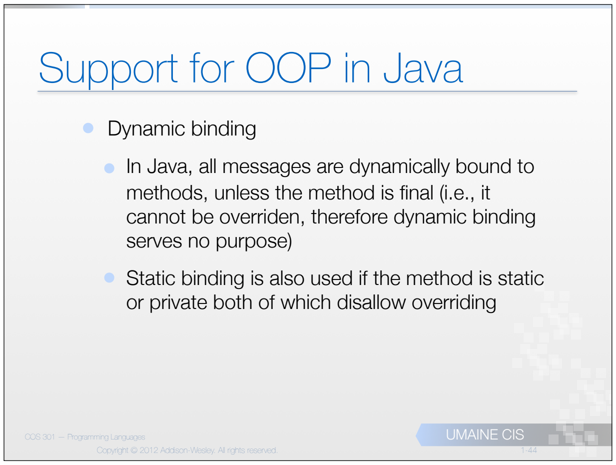


Support for OOP in Java

- Inheritance
  - Single inheritance supported only, but there is an abstract class category that provides some of the benefits of multiple inheritance (**interface**)
  - An interface can include only method declarations and named constants, e.g.,

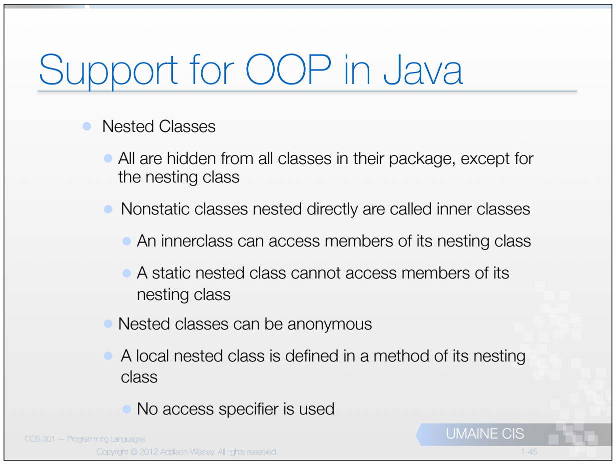
```
public interface Comparable <T> {
        public int comparedTo (T b);
}
```

  - Methods can be final (cannot be overriden)



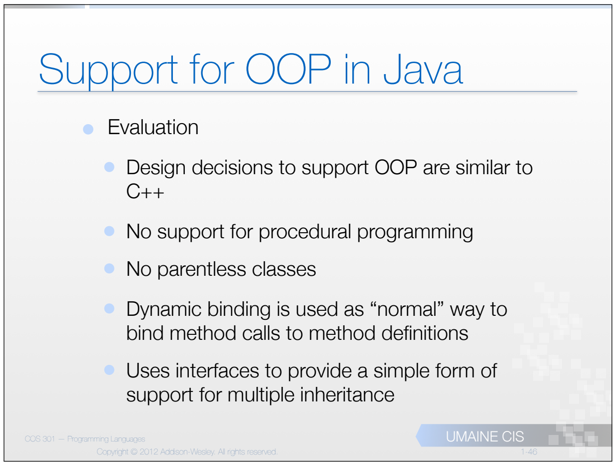
Support for OOP in Java

- Dynamic binding
  - In Java, all messages are dynamically bound to methods, unless the method is final (i.e., it cannot be overriden, therefore dynamic binding serves no purpose)
  - Static binding is also used if the method is static or private both of which disallow overriding



Support for OOP in Java

- Nested Classes
  - All are hidden from all classes in their package, except for the nesting class
  - Nonstatic classes nested directly are called inner classes
    - An innerclass can access members of its nesting class
    - A static nested class cannot access members of its nesting class
  - Nested classes can be anonymous
  - A local nested class is defined in a method of its nesting class
    - No access specifier is used



Support for OOP in Java

- Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming
  - No parentless classes
  - Dynamic binding is used as “normal” way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance

Support for OOP in C#

- General characteristics
    - Support for OOP similar to Java
    - Includes both classes and structs
    - Classes are similar to Java's classes
    - structs are less powerful stack-dynamic constructs (e.g., no inheritance)

Support for OOP in C#

- Inheritance
    - Uses the syntax of C++ for defining classes
    - A method inherited from parent class can be replaced in the derived class by marking its definition with new
    - The parent class version can still be called explicitly with the prefix base:

      base.Draw()

Support for OOP in C#

- Dynamic binding
    - To allow dynamic binding of method calls to methods:
        - The base class method is marked virtual
        - The corresponding methods in derived classes are marked override
    - Abstract methods are marked abstract and must be implemented in all subclasses
    - All C# classes are ultimately derived from a single root class, Object

---

Support for OOP in C#

- Nested classes
    - A C# class that is directly nested in a nesting class behaves like a Java static nested class
    - C# does not support nested classes that behave like the non-static classes of Java

Support for OOP in C#

- Evaluation
  - C# is a relatively recently designed C-based OO language
  - The differences between C#'s and Java's support for OOP are relatively minor

---

Support for OOP in Ada

- General Characteristics
  - OOP was one of the most important extensions to Ada 83
  - Encapsulation container is a package that defines a tagged type
  - A tagged type is one in which every object includes a tag to indicate during execution its type (the tags are internal)
  - Tagged types can be either private types or records
  - No constructors or destructors are implicitly called

Support for OOP in Ada 95 (continued)

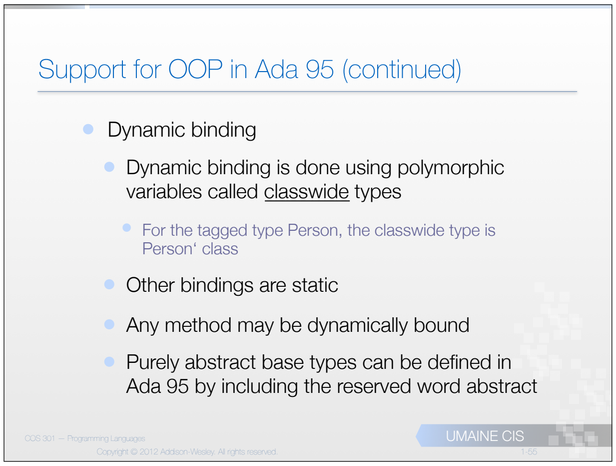
- Inheritance
  - Subclasses can be derived from tagged types
  - New entities are added to the inherited entities by placing them in a record definition
  - All subclasses are subtypes
  - No support for multiple inheritance
    - A comparable effect can be achieved using generic classes

---

Example of a Tagged Type

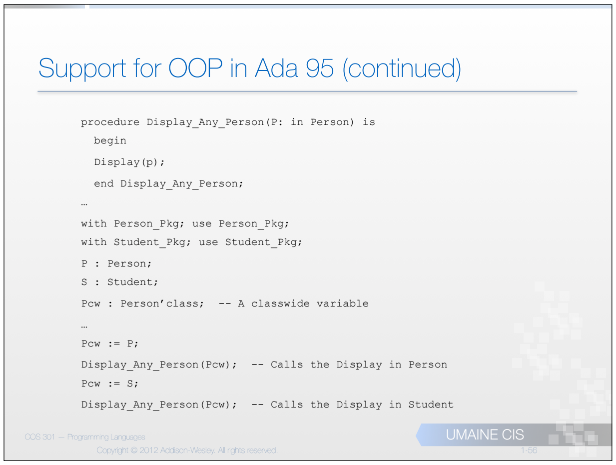
```
package Person_Pkg is
  type Person is tagged private;
  procedure Display(P : in out Person);
  private
    type Person is tagged
      record
        Name : String(1..30);f
        Address : String(1..30);
        Age : Integer;
      end record;
end Person_Pkg;
with Person_Pkg; use Person_Pkg;
 package Student_Pkg is
   type Student is new Person with
     record
       Grade_Point_Average : Float;
       Grade_Level : Integer;
     end record;
   procedure Display (St: in Student);
 end Student_Pkg;

// Note: Display is being overridden from Person_Pkg
```



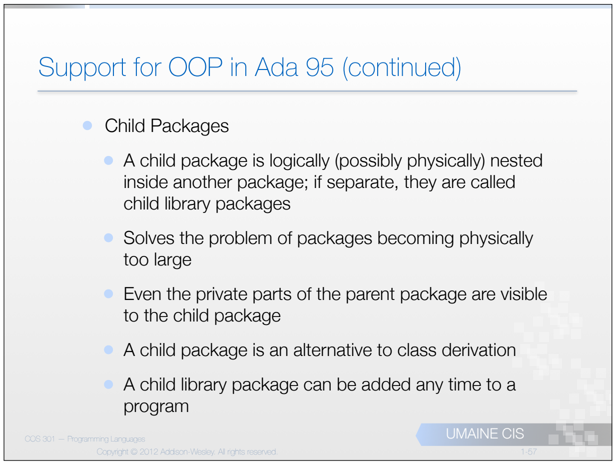
Support for OOP in Ada 95 (continued)

- Dynamic binding
  - Dynamic binding is done using polymorphic variables called <u>classwide</u> types
    - For the tagged type Person, the classwide type is Person' class
  - Other bindings are static
  - Any method may be dynamically bound
  - Purely abstract base types can be defined in Ada 95 by including the reserved word abstract



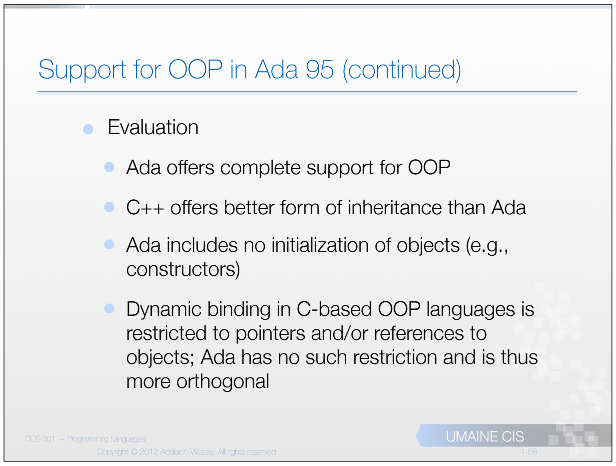
Support for OOP in Ada 95 (continued)

```
procedure Display_Any_Person(P: in Person) is
  begin
  Display(p);
  end Display_Any_Person;
…
with Person_Pkg; use Person_Pkg;
with Student_Pkg; use Student_Pkg;
P : Person;
S : Student;
Pcw : Person’class;  -- A classwide variable
…
Pcw := P;
Display_Any_Person(Pcw);  -- Calls the Display in Person
Pcw := S;
Display_Any_Person(Pcw);  -- Calls the Display in Student
```



Support for OOP in Ada 95 (continued)

- Child Packages
  - A child package is logically (possibly physically) nested inside another package; if separate, they are called child library packages
  - Solves the problem of packages becoming physically too large
  - Even the private parts of the parent package are visible to the child package
  - A child package is an alternative to class derivation
  - A child library package can be added any time to a program



Support for OOP in Ada 95 (continued)

- Evaluation
  - Ada offers complete support for OOP
  - C++ offers better form of inheritance than Ada
  - Ada includes no initialization of objects (e.g., constructors)
  - Dynamic binding in C-based OOP languages is restricted to pointers and/or references to objects; Ada has no such restriction and is thus more orthogonal

Support for OOP in Ruby

- General Characteristics
  - Everything is an object
  - All computation is through message passing
  - Class definitions are executable, allowing secondary definitions to add members to existing definitions
  - Method definitions are also executable
  - All variables are type-less references to objects
  - Access control is different for data and methods
    - It is private for all data and cannot be changed
    - Methods can be either public, private, or protected
    - Method access is checked at runtime
  - Getters and setters can be defined by shortcuts

Support for OOP in Ruby

- Inheritance
  - Access control to inherited methods can be different than in the parent class
  - Subclasses are not necessarily subtypes
  - Mixins can be created with modules, providing a kind of multiple inheritance
- Dynamic Binding
  - All variables are typeless and polymorphic
- Evaluation
  - Does not support abstract classes
  - Does not fully support multiple inheritance
  - Access controls are weaker than those of other languages that support OOP

Implementing OO

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

Instance Data Storage

- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR -> child's CIR
- Because CIR is static, access to all instance variables is done as it is in records: Efficient

Dynamic Binding of

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called virtual method tables (*vtable*)
  - Method calls can be represented as offsets from the beginning of the vtable

Summary

- OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding
- Major design issues: exclusivity of objects, subclasses and subtypes, type checking and polymorphism, single and multiple inheritance, dynamic binding, explicit and implicit de-allocation of objects, and nested classes
- Smalltalk is a pure OOL
- C++ has two distinct type systems (hybrid)
- Java is not a hybrid language like C++; it supports only OOP
- C# is based on C++ and Java
- Ruby is a relatively recent pure OOP language; provides some new ideas in support for OOP
- Implementing OOP involves some new data structures