Review of OO concepts

Objects bundle data with the operations on the data

Objects provide encapsulation

Class-based and prototype-based languages

Code reuse through inheritance
Inheritance and subtyping

OO languages provide *subtype polymorphism*
- A instance of a subtype can be used anywhere an instance of a supertype can be used

**Requires:** any operation on supertype must be supported on a subtype

- *Inheritance* achieves this:
  - a subclass is also subtype (usually)
  - a subtype should support *more* operations than a supertype
Often, subtyping implies an “is a” relationship

- A Penguin is a Bird

But:

- A Stack is a List
  - However, a Stack supports fewer operations than List
  - A Stack is not a subtype of List
Stack and List

Stack <: List?
- a stack is a list, but with some restrictions

List <: Stack?
- a list has more operations than a stack – can use a stack as a list

“is a” isn’t always the “right” way to decide when to use inheritance

Many OO programs overuse inheritance

Often better to use containment
- Stack should contain a List—can hide operations on the List
- Stack should not be a subtype of List
Java collections library

Stack is a subtype of List
- Broken: allows a Stack to be used in a non-stack-like way

Unmodifiable collections
- throws UnsupportedOperationException if try to write to the collection
Virtual methods

In C++ methods may be declared *virtual*.
In Java, all (non-static) methods are virtual.

Which method body to run depends on *receiver’s* run-time class

class A { void m(); }
class B extends A { void m(); }

A x = new B();
x.m(); // invokes B.m()
A y = new A();
y.m(); // invokes A.m()
Self (aka this)

Virtual methods have a formal parameter for the method *receiver*
- Type of the receiver is the enclosing class

In C++, Java, C#, the receiver is an implicit parameter named `this`
In Smalltalk, Ruby, receiver is an implicit parameter named `self`

In Python and some other languages, receiver is explicitly named.
Fields are not virtual

In C++ and Java, field access is *not* virtual
- which field to use depends on the static type of the target

class A { int f; }
class B extends A { int f; }

A x = new B();
x.f;   // accesses A.f
A y = new A();
y.f;   // accesses A.f
B z = new B();
z.f;   // accesses B.f
Constructors

Constructors are procedures that initialize objects.

Job of constructor is to establish the *object invariant*.

Invariant can be assumed on entry to any non-private method and must be reestablished on return.

Constructor for a subclass invokes constructors for superclasses, passing arguments up to initialize fields.
class C {
    final int f;
    C() { init(); f = 1; }
    void init() { }
}

class D extends C {
    D() { super(); }
    void init() { println(f); }
}

new D()  // prints what?
Java constructor anomaly

class C {
    final int f;
    C() { init(); f = 1; }
    void init() { }
}

class D extends C {
    D() { super(); }
    void init() { println(f); }
}

new D()  // prints 0
Objects provide *encapsulation*

- Can hide implementation details
- Use the object only through a narrow interface
Visibility

*Public* members are visible everywhere

*Private* members are visible only to members of the same class

*Protected* members are visible to the current class and to subclasses only

Java:
- package-scoped (default) and *also protected* members are visible to other classes in the same package
Problem

Inheritance violates encapsulation!

Subclasses know implementation details of superclasses

But, also *requires* subclasses to know the implementation details of their superclasses

Class provides two interfaces:
- one interface for external users
- one interface for subclasses
java.lang.Properties inherits from java.lang.Hashtable

“Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a “compromised” Properties object that contains a non-String key or value, the call will fail.”
Abstract classes and methods

Methods can be declared abstract (pure virtual)

- Java: abstract void m();
- C++: virtual void m() = 0;

Abstract methods don’t have a body.
A class with an abstract method must also be abstract.
An abstract class cannot be instantiated.
=> Non-abstract classes must override abstract methods they inherit.
Multiple inheritance

A class may have more than one superclass
Effect is still as if members are copied
Java supports only multiple inheritance of *interfaces*
C++ supports arbitrary multiple inheritance
  - causes all sorts of complications
Other languages have restricted MI to simplify semantics
MI problem 1: ambiguous methods

class A { void m() }
class B { void m() }
class C extends A, B { }

C inherits an m() from A and an m() from B.

C++: caller must specify which to use:
- C *x = new C;
- x->A::m();
- x->B::m();
MI problem 1: ambiguous methods

interface A { void m() }
interface B { void m() }
class C implements A, B { void m() { S } }

In Java, multiple interface inheritance is okay
Don’t inherit more than one implementation
C is required to implement m()
class A { int f; }
class B { int f; }
class C extends A, B { }

C inherits A.f and B.f.

This is okay because field access is nonvirtual.

Only issue is in C, accessing this.f is ambiguous.

- this->A::f vs. this->B::f
MI problem 2: diamonds

class A { int f; }
class B extends A { }
class C extends A { }
class D extends B, C { }

Does D have one copy of f or two?
MI problem 2: diamonds

C++ supports virtual inheritance

class A { int f; }
class B : virtual A { }
class C : virtual A { }
class D : B, C { }

D shares one copy of A’s members.

Annoying problem: B and C are declared to use virtual inheritance. Have to anticipate that B and C will be multiply inherited.
Java interfaces

Java gets around this problem by only allowing multiple inheritance of interfaces.

Interface = class with no fields and where all methods are public, abstract.

Multiple inheritance of interfaces
Single inheritance of classes
=> only one copy of every field
Java interfaces

All interface methods are abstract
- must be implemented by any class implementing the interface

Separation of specification and implementation
What if we allow non-abstract methods in interfaces?

Can’t do this in Java, but can in Smalltalk, Fortress

- called traits

Trait = a class without fields

Single inheritance of classes
Multiple inheritance of traits ("trait composition")
Mixins

Another variation of multiple inheritance
- CLOS, Jigsaw, Mixgen, can simulate in C++

Parameterized superclass

```java
mixin Colored<T> extends T {
    String color;
}
```

```java
class Carrot extends Colored<Vegetable> { ... }
```
Scala has a hybrid of mixins and traits

Called **traits**

But can have fields

Linear ordering of superclasses like mixins
trait Colored {
    def color: String
    def hasSameColor(x: Colored) = color == x.color
}

class Banana extends Fruit with Colored {
    def color = “yellow”
}

(new Banana) hasSameColor (new Grapefruit)
Virtual inheritance and constructors

```cpp
class A {  int f;  A(int x) : f(x); }  
class B : virtual A {  B() : A(1); }  
class C : virtual A {  C() : A(2); }  
class D : B, C { }  
```

B’s constructor invokes A’s
C’s constructor invokes A’s
D’s constructor invokes B’s and C’s

Is A’s constructor called twice? Possibly with different args?
Virtual inheritance and constructors

class A {  int f;  A(int x) : f(x);  }
class B : virtual A {  B() : A(1);  }
class C : virtual A {  C() : A(2);  }
class D : B, C {
  D() : A(0), B(), C();
}

If D inherits from a virtual base class A it must explicit invoke A’s constructor. B and C will invoke A’s constructor only when new B and new C are done, not new D.

Ensures A() is invokes only once.
Static semantics

Recall rule for function calls:

\[
A \vdash e_0 : (T_1, \ldots, T_n) \Rightarrow T \quad A \vdash e_i : T_i
\]

\[
\begin{array}{c}
A \vdash e_0(e_1, \ldots, e_n) : T \\
\end{array}
\]

Actually, a bit more complicated:

\[
A \vdash e_0 : (T_1, \ldots, T_n) \Rightarrow T \quad A \vdash e_i : S_i \quad S_i <: T_i
\]

\[
\begin{array}{c}
A \vdash e_0(e_1, \ldots, e_n) : T \\
\end{array}
\]
New rule for function calls:

\[
A \vdash e_0 : (T_1, \ldots, T_n) \Rightarrow T \quad A \vdash e_i : S_i \quad S_i <: T_i
\]

\[
A \vdash e_0(e_1, \ldots, e_n) : T
\]
Static semantics

New rule for function calls:

\[ A \vdash e_0 : (T_1, \ldots, T_n) \Rightarrow T \quad A \vdash e_i : S_i \quad S_i <: T_i \]

\[ \frac{}{A \vdash e_0(e_1, \ldots, e_n) : T} \]

Method calls:

\[ A \vdash e_0 : C \quad A \vdash e_i : T_i \quad S_i <: T_i \quad C <: B \quad \text{B has method} \]

\[ T \text{ m}(T_1, \ldots, T_n) \]

\[ \frac{}{A \vdash e_0.m(e_1, \ldots, e_n) : T} \]
Static semantics for field accesses

\[ A \vdash e_0 : C \quad C <: B \quad B \text{ has field } T \ f \]

\[ A \vdash e_0.f : T \]
Method overloading

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

c c = new C();
c.m("hello"); // calls m(String)
c.m(1); // calls m(int);
c.m(new ArrayList()); // calls m(Object)
c.m(null); // ?
Method overloading

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

C c = new C();
c.m("hello");  // calls m(String)
c.m(1);         // calls m(int);
c.m(new ArrayList()); // calls m(Object)
c.m(null);      // error! m(String) or m(Object)?
Method overloading

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

c = new C();
c.m("hello"); // calls m(String)
c.m(1); // calls m(int);
c.m(new ArrayList()); // calls m(Object)
c.m((Object) null); // calls m(Object)
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x);  // overrides C.m(String)
    void m(Object x);  // overrides C.m(Object)
    void m(List x);   // a new method
}
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x);   // overrides C.m(String)
    void m(Object x);  // overrides C.m(Object)
    void m(List x);    // a new method
}

C c = new D();
c.m(new ArrayList());    // calls what?
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x);    // overrides C.m(String)
    void m(Object x);    // overrides C.m(Object)
    void m(List x);      // a new method
}

C c = new D();
c.m(new ArrayList());    // calls D.m(Object)
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x);  // overrides C.m(String)
    void m(Object x);  // overrides C.m(Object)
    void m(List x);    // a new method
}

D d = new D();
d.m(new ArrayList());  // calls D.m(List)
If language supports overloading, method lookup is more complicated.

Don’t just look for method with the same name.

Find for all methods with same name whose parameter types are supertypes of the actual argument types.

Then, select the *most specific method*, or report that the call is ambiguous.
Most specific method

In Java (simplified):

M1 defined in C1 is *more specific* than M2 in C2 if:

- the formal params of M1 are subtypes of the formal params of M2
  - m(Integer) is more specific than m(Number)
- or, the formal param types are the same, but C1 is a subtype of C2
  - C1.m(T) is more specific than C2.m(T)

There may not be a single most specific method for a given call.

=> The call is ambiguous.
Conformance checking

Compiler needs to ensure a subclass is a *subtype* of its base class.

If a subclass overrides method m, argument types and return types must be compatible.

Need to ensure all abstract methods are implemented.

Need to check access flags, exceptions also
- e.g., cannot override a public method with a private method
- e.g., method in subclass cannot throw *more* exceptions than method in superclass
Method conformance

Without method overloading:
- If a subclass declares a method $M_1$ with the same name as a superclass method $M_2$, $M_1$ must also override $M_2$
- $T \ m(T_1, \ldots, T_n)$ overrides $S \ m(S_1, \ldots, S_k)$ if:
  - $n = k$
  - $T <: S$ — covariant return types
  - $S_i <: T_i$ — contravariant argument types
Covariant return types - ok

```java
class C {
    Number m() { return 3.14; }
}

class D extends C {
    Integer m() { return 1; }
}

C c = new D();
Number x = c.m();  // ok!
```
class C {
    Integer m() { return 0; }
}

class D extends C {
    Number m() { return 2.718281828; }
}

C c = new D();
Integer x = c.m();    // uh-oh!
Contravariant argument types - ok

class C {
    void m(Integer x);
}

class D extends C {
    void m(Number x);
}

C c = new D();
c.m(1);    // ok, 1 is also a Number
Covariant argument types - wrong

```java
class C {
    void m(Number x);
}

class D extends C {
    void m(Integer x);
}

C c = new D();
c.m(1.618);  // uh-oh! 1.618 is a Number, but not an Integer
```
**Method conformance**

*With* method overloading:
- If a subclass declares a method M1 with the same *signature* as a superclass method M2, M1 must also *override* M2
- signature = name + parameter types
- T m(T1, ..., Tn) overrides S m(S1, ..., Sk) if:
  - n = k
  - T <: S  – covariant return types
- Argument types must be the same because changing the argument types *overloads* the method
interface Set {
    void union(Set s);
}

This is wrong:
class BitSet implements Set {
    int bits;
    public void union(BitSet s) { bits |= s.bits; } // compiler error!
}

This compiles:
class BitSet implements Set {
    public void union(Set s) { if (s instanceof BitSet) { ... } }
}
Binary methods

Typing rules prevent subclasses from assuming that arguments are of the same class.

BitSet.union cannot assume it will be passed a BitSet since the caller might be using the BitSet at type Set.
Implementing objects and records
Records

aka structs

struct S {
    int x;
    double y;
}

Thursday, April 1, 2010
Records

```c
void f() {
    struct S s; // declares a local variable of type S
    s.x = 0;
    s.y = 1.0;
}
```

equivalent to expansion of the struct:

```c
void f() {
    int s$x;
    double s$y;
    s$x = 0;
    s$y = 1.0;
}
```
Allocating records on the heap

```c
void f() {
    struct S *s = new S;
    s->x = 0;
    s->y = 1.0;
    delete s;
}
```

- allocates a struct on the heap
- saves a pointer to the struct on the stack
- must dereference the pointer to access its fields

- in C++, should explicitly free the memory with delete
- in a language like Java, GC will free the memory
void f() {
    struct S *s = new S;
    s->x = 0;
    s->y = 1.0;
    delete s;
}

Thursday, April 1, 2010
class C {
    int x;
    double y;
    virtual void m();
}

void f() {
    C *c = new C;
    c->x = 0;
    c->y = 1.0;
    delete c;
}

Objects have an object header
class C {
    int x;
    double y;
    void m();
}

void f() {
    C c = new C();
    c.x = 0;
    c.y = 1.0;
}
class C {
    int x;
    double y;
    void m();
}

void f() {
    C c = new C();
    c.x = 0;
    c.y = 1.0;
}

**Note**: object can outlive the method that created it since it lives in the heap.
Objects on the stack

In C++, can also allocate objects on the stack
We will ignore this.
Inheritance

```java
class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new D();
}
```

With single inheritance, fields of the subclass are appended to the object
Inheritance

class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new D();
}

Note: prefix of a D object looks exactly like a C object.
Subtyping

class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new C();
    C d = new D();
}

Note: prefix of a D object looks exactly like a C object. This allows a D to be used as a C.
Method dispatch

class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new C();
    C d = new D();
}
Method dispatch

\( x.m() \)

implemented as:

\[
t = *x; \quad \text{// dereference } x \text{ to get base} \\
\quad \text{// address of dispatch table} \\
p = *(t + m\_offset); \quad \text{// load the address of the} \\
\quad \text{// method’s code} \\
p(); \quad \text{// invoke the method}
\]
class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
    void m();
}

void f() {
    C c = new C();
    C d = new D();
}

To implement method override, just change address of code in dispatch table.
Method dispatch

Code generated for calling an overridden method is exactly the same.

 Caller doesn’t know that the method is overridden.

 Caller doesn’t even know the exact run-time class of the receiver.
Dynamic casts and instanceof

Can ask for run-time type of an object.

```java
e instanceof C
(C) e
```

Simple implementation:
- just add a hidden virtual method that returns a representation of the type
Multiple inheritance

Shared vs. non-shared

- given a diamond:
  - class A
  - class B extends A
  - class C extends A
  - class D extends B, C
- does D contain one copy of A (shared) or two (non-shared)?

- C++ uses virtual inheritance to make A shared
- otherwise non-shared
class A { int w; }
class B : A { int x; }
class C : A { int y; }
class D : B, C { int z; }

Note, D has two copies of w
Also two headers.
class A { int w; }
class B : virtual A { int x; }
class C : virtual A { int y; }
class D : B, C { int z; }

Note B, C, D have pointers to the “A subobject” of the object.
Interfaces

interface I {
    void n();
    void m();
}

interface J {
    void m();
    void p();
}

class C implements I, J {
    int x;
    void p() { ... }
    void n() { ... }
    void m() { ... }
}

class D implements I {
    void m() { ... }
    void n() { ... }
}
Interfaces

interface I {
    void n();
    void m();
}

interface J {
    void m();
    void p();
}

class C implements I, J {
    int x;
    void p() { ... }
    void n() { ... }
    void m() { ... }
}

class D implements I {
    void m() { ... }
    void n() { ... }
}

I x = new D();
x.m();
J y = new C();
y.m();

How to do method dispatch when table entries for method m don’t line up?
Interface dispatch

One approach, used by C++ to implement MI

Multiple dispatch tables per type
- one per superclass
- space inefficient
- relies on knowing all superclasses at compile-time (not possible in Java with dynamic class loading)
Problem:

dispatch table offset for a method is determined by the class hierarchy

a given method (say m) may be at different offsets in the dispatch table because it was not introduced by a common superclass
Interface dispatch

For each interface \( I \) a class \( C \) implements, maintain \( itable(I,C) \), a dispatch table for \( C \), restricted to methods of \( I \).

To dispatch to a method \( m \) of \( I \), first find \( itable(I,C) \).
- can be located through a global hash table

Then locate the code address with a normal dispatch table lookup.
interface I {
    void n();
    void m();
}

interface J {
    void m();
    void p();
}

class C implements I, J {
    int x;
    void p() { ... }
    void n() { ... }
    void m() { ... }
}

class D implements I {
    void m() { ... }
    void n() { ... }
}