Object implementation
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 implemented using a dispatch table, accessed through the object header.
Method dispatch

\[ x.m() \]

implemented as:

\[ t = *x; \quad \text{// dereference } x \text{ to get base} \]
\[ \quad \text{// address of dispatch table} \]
\[ p = *(t + m_{\text{offset}}); \quad \text{// load the address of the} \]
\[ \quad \text{// method’s code} \]
\[ (*p)(); \quad \text{// invoke the method} \]
Method override

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;
}
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.
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
Non-virtual MI

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.
Virtual MI

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() { ... }
}

*p is address of dispatch table
Fat pointers

Another implementation of interfaces:

Before every pointer of interface type to an object, add a pointer to a dispatch vector for that type

```cpp
l p = new C();
```

- vtbl for I
- p
- header
- x

- ptr to code for C.p
- ptr to code for C.n
- ptr to code for C.m
- ptr to code for C.m
Dynamic casts and instanceof

Can ask for run-time type of an object.

e instanceof C
  (C) e

Simple implementation:
• just add a hidden virtual method that returns a representation of the type
Method overloading
Method overloading

class C {
    void m(Bird x);
    void m(Mammal x);
    void m(Object x);
}

C c = new C();
c.m(new Cow());  // calls m(Bird)
c.m(new Dog());  // calls m(Mammal);
c.m(new Cabbage());  // calls m(Object)
c.m(null);  // calls what?
Method overloading

class C {
    void m(Bird x);
    void m(Mammal x);
    void m(Object x);
}

c C c = new C();
c.m(new Cow()); // calls m(Bird)
c.m(new Dog()); // calls m(Mammal);
c.m(new Cabbage()); // calls m(Object)
c.m(null); // error! is null Bird, Mammal, or Object?
Method overloading vs. overriding

class C {
    void m(Bird x);
    void m(Mammal x);
    void m(Object x);
}

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

class C {
    void m(Bird x);
    void m(Mammal x);
    void m(Object x);
}

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

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

class C {
    void m(Bird x);
    void m(Mammal x);
    void m(Object x);
}

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

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

class C {
    void m(Bird x);
    void m(Mammal x);
    void m(Object x);
}

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

D d = new D();
d.m(new Cabbage());    // calls D.m(Vegetable)
Method lookup

• 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(Cow) is more specific than m(Mammal)
  • 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
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 M1 with the same name as a superclass method M2, M1 must also override M2
  • $T \; m(T_1, \ldots, T_n)$ overrides $S \; m(S_1, \ldots, S_k)$ if:
    • $n = k$
    • $T <: S$ – covariant return types
    • $S_i <: T_i$ – contravariant argument types
class C {
    Bird m() { return new Penguin(); }
}

class D extends C {
    Duck m() { return new Duck(); }
}

C c = new D();
Bird x = c.m(); // ok!
Contravariant return types - wrong

class C {
    Penguin m() { return new Penguin(); }
}

class D extends C {
    Bird m() { return new Duck(); }
}

C c = new D();
Penguin x = c.m(); // uh-oh!
class C {
    void m(Duck x);
}

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

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

class C {
    void m(Bird x);
}

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

C c = new D();
c.m(new Penguin());  // uh-oh! a Bird, but not an Duck
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
  • \( T \ m(T_1, \ldots, T_n) \) overrides \( S \ m(S_1, \ldots, S_k) \) if:
    • \( n = k \)
    • \( T <: S \) – covariant return types

• Argument types must be the same because changing the argument types **overloads** the method
Binary methods

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 {
    int bits;
    public void union(Set s) {
        if (s instanceof BitSet) { bits |= ((BitSet) s).bits; } else ... }
}
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.
Questions?