CSE 3302

Lecture 6: Even still more objects
14 September 2010

Nate Nystrom
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 Cowbird());       // 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 = new C();
c.m(new Cowbird());    // 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.
Multimethods

• Methods: use the dynamic type of the receiver to choose which method to invoke
  • **single dispatch**

• Multimethods: use the dynamic type of all arguments to choose which method to invoke
  • **multiple dispatch**

• Languages that support multimethods:
  • CLOS (Common Lisp Object System)
  • Cecil
  • MultiJava
  • C# 4.0 (although MS won’t admit it)
Multimethods in C# 4.0

Use the new **dynamic** type!

class Shapes {
    static boolean intersect(Circle c1, Circle c2) {
        return dist(c1.center, c2.center) <= c1.radius + c2.radius;
    }
    static boolean intersect(Rect r1, Rect r2) { ... }
    static boolean intersect(Rect r, Circle c) { ... }
    static boolean intersect(Circle c, Rect r) { ... }
}

dynamic c1 = new Circle(0, 0, 1);
dynamic c2 = new Circle(0.5, -0.5, 2);

Shapes.intersect(c1, c2);
// Uses dynamic type of c1, c2 to select which method to call.
// Throws exception if none match
Visitors

- Can achieve **double dispatch** with the **visitor** design pattern
Abstract syntax trees

abstract class Exp {
    abstract int eval();
}
class Add {
    Exp x, y;
    int eval() { return x.eval() + y.eval(); }
}
class Int extends Exp {
    int value;
    int eval() { return value; }
}
Let’s add a print operation

abstract class Exp {
    abstract int eval();
    abstract void print();
}
class Add {
    Exp x, y;
    int eval() { return x.eval() + y.eval(); } 
    void print() {
        println("(");
        x.print();
        println(" + ");
        y.print();
        println("")
    }
}
class Int extends Exp {
    int value;
    int eval() { return value; } 
    void print() { println(value); } 
}
Problem

• Have to change a bunch of existing classes to add new operations.

• In many kinds of programs, type of data doesn’t change often, but new operations are added all the time.

• Solution: the visitor design pattern
Visitors

• Represent each data type as a class
• Represent each operation as a class

• **Double dispatch** on the data type and the operation
abstract class Exp {
    abstract <T> accept(Visitor<T> v);
}

class Add extends Exp {
    Exp x, y;
    <T> accept(Visitor<T> v) { return v.visit(this); }
}

class Int extends Exp {
    int value;
    <T> accept(Visitor<T> v) { return v.visit(this); }
}
abstract class Visitor<T> {
    abstract T visit(Add n);
    abstract T visit(Int n);
}
class Eval extends Visitor<Integer> {
    Integer visit(Add n) {
        return n.x.accept(this) + n.y.accept(this);
    }
    Integer visit(Int n) { return n.value; }
}

ast.accept(new Eval());
Print visitor

class Print extends Visitor<Void> { 
    Void visit(Add n) { 
        println("(");
        n.x.accept(this);
        println(" + ");
        n.y.accept(this);
        println(")");
        return null;
    }
    Void visit(Int n) { 
        println(n.value);
        return null;
    }
}

ast.accept(new Print());
Compilers and interpreters

Typical compiler or interpreter structure:

• Abstract syntax trees represent the program
  • Constructed by a parser
    • converts sequence of bytes into an AST

• Multiple passes over the ASTs implemented using visitors
  • Semantic checking
    • check types, exceptions, unreachable code, uninitialized variables, ...
  • Optimizations (lots of these: 10, 20 or more)
  • Code generation (compilers only, output target code)
  • Evaluation (interpreters only)
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 $M_1$ 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

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
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(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
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?