CSE 3302

Lecture 4: More objects
7 September 2010

Nate Nystrom
University of Texas at Arlington
Administration

- Join the mailing list:
  - http://groups.google.com/group/uta-cse-3302

- HW1 due Thursday 9/9
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** (subclassing) achieves this:
  • a subclass is also subtype (usually)

• a subtype should support **more** operations than a supertype
Subtyping vs. subclassing

• Subclasses are *usually* also subtypes

• But:
  • In some languages, can choose to subclass without subtyping
  • Could allow the subclass to be a subtype, but choose not to

• C++: private inheritance
Private inheritance

In C++, can inherit privately:

class A { }
class B : private A { }

A *x = new B(); // illegal!

// B is not a subtype of A
Why private inheritance?

Want to reuse code but violate “is a” relationship

class Engine { void start() { ... } }
class Car : private Engine {
    using Engine::start;
}

Usually better to just use containment:

class Car {
    Engine e;
    void start() { e.start(); }
}
Subtyping vs. subclassing

Can also be a subtype without being a subclass

Primitives:
• int <: int

Generics (Scala):
• List[Robin] <: List[Bird]

Generics (Java):
• List<Bird> <: List<? extends Bird>
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, the receiver is an implicit parameter named **self**

• In Python and some other languages, the 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

```java
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 Link {
    Link next;
    Link prev;
    int data;
}

class List {
    Link nil;
    // establish the object invariant:
    // nil is a dummy node between tail and head
    List() { nil = new Link();
        nil.next = nil.prev = nil; }

    Link head() { return nil.next; }
    Link tail() { return nil.prev; }
}
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 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()
MI problem 1(b): ambiguous fields

- 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
Traits

• 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

- mixin Colored<T> extends T {
  - String color;
  - }

- class Carrot extends Colored<Vegetable> { ... }
Scala

- Scala has a hybrid of mixins and traits
- Called **traits**
- But can have fields
- Linear ordering of superclasses like mixins
Traits

• 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

- 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 explicitly 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.
MI problem 2: diamonds

• With nonvirtual inheritance:

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

• D has two copies of A’s members, those from B and those from C.
• Uses of D (including this) must specify which copy of f to access, using d->B::f and d->C::f.
Implementing objects and records
Records

aka structs

struct S {
  int x;
  double y;
}

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:

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;
}
Objects

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

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

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

x.m()

implemented as:

t = *x;              // dereference x to get base
                   // address of dispatch table
p = *(t + m_offset);   // load the address of the
                   // method’s code
(*p)();            // 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();
}

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.

\[ e \text{ instanceof } C \]

\[(C) \ e\]

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
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?
Questions?