

CIS 500
Software Foundations
Fall 2006

November 20

On to Objects

A Change of Pace

We've spent the semester developing tools for defining and reasoning about a variety of programming language features. Now it's time to *use* these tools for something more ambitious.

Case study: object-oriented programming

Plan:

1. Identify some characteristic “core features” of object-oriented programming
2. Develop two different analyses of these features:
 - 2.1 A *translation* into a lower-level language
 - 2.2 A *direct*, high-level formalization of a simple object-oriented language (“Featherweight Java”)

The Translational Analysis

Our first goal will be to show how many of the basic features of object-oriented languages

dynamic dispatch

encapsulation of state

inheritance

late binding (this)

super

can be understood as “derived forms” in a lower-level language with a rich collection of primitive features:

(higher-order) functions

records

references

recursion

subtyping

The Translational Analysis

For simple objects and classes, this translational analysis works very well.

When we come to more complex features (in particular, classes with `this`), it becomes less satisfactory, leading us to the more direct treatment in the following chapter.

Concepts

The Essence of Objects

What “is” object-oriented programming?

The Essence of Objects

What “is” object-oriented programming?

A precise definition has been the subject of debate for decades. Such arguments are always inconclusive and seldom interesting.

The Essence of Objects

What “is” object-oriented programming?

A precise definition has been the subject of debate for decades. Such arguments are always inconclusive and seldom interesting.

However, it is easy to identify some core features that are shared by most OO languages and that, together, support a distinctive and useful programming style.

Dynamic dispatch

Perhaps the most basic characteristic of object-oriented programming is *dynamic dispatch*: when an operation is invoked on an object, the ensuing behavior depends on the object itself, rather than being fixed (as when we apply a function to an argument).

Two objects of the *same type* (i.e., responding to the same set of operations) may be implemented internally in *completely different* ways.

Example (in Java)

```
class A {
    int x = 0;
    int m() { x = x+1; return x; }
    int n() { x = x-1; return x; }
}

class B extends A {
    int m() { x = x+5; return x; }
}

class C extends A {
    int m() { x = x-10; return x; }
}
```

Note that `(new B()).m()` and `(new C()).m()` invoke completely different code!

Encapsulation

In most OO languages, each object consists of some internal state *encapsulated* with a collection of method implementations operating on that state.

- ▶ state directly accessible to methods
- ▶ state inaccessible from outside the object

Encapsulation

In Java, encapsulation of internal state is optional. For full encapsulation, fields must be marked `protected`:

```
class A {
    protected int x = 0;
    int m() { x = x+1; return x; }
    int n() { x = x-1; return x; }
}

class B extends A {
    int m() { x = x+5; return x; }
}

class C extends A {
    int m() { x = x-10; return x; }
}
```

The code `(new B()).x` is not allowed.

Side note: Objects vs. ADTs

The encapsulation of state with methods offered by objects is a form of *information hiding*.

A somewhat different form of information hiding is embodied in the notion of an *abstract data type* (ADT).

Side note: Objects vs. ADTs

An ADT comprises:

- ▶ A *hidden* representation type X
- ▶ A collection of operations for creating and manipulating elements of type X .

Similar to OO encapsulation in that only the operations provided by the ADT are allowed to directly manipulate elements of the abstract type.

But *different* in that there is just one (hidden) representation type and just one implementation of the operations — no dynamic dispatch.

Both styles have advantages.

Caveat: In the OO community, the term “abstract data type” is often used as more or less a synonym for “object type.” This is unfortunate, since it confuses two rather different concepts.

Subtyping and Encapsulation

The “type” (or “interface” in Smalltalk terminology) of an object is just the set of operations that can be performed on it (and the types of their parameters and results); it does not include the internal representation.

Object interfaces fit naturally into a subtype relation.

An interface listing more operations is “better” than one listing fewer operations.

This gives rise to a natural and useful form of polymorphism: we can write one piece of code that operates uniformly on any object whose interface is “at least as good as **I**” (i.e., any object that supports at least the operations in **I**).

Example

```
// ... class A and subclasses B and C as above...
```

```
class D {  
    int p (A myA) { return myA.m(); }  
}
```

```
...
```

```
D d = new D();  
int z = d.p (new B());  
int w = d.p (new C());
```

Inheritance

Objects that share parts of their interfaces will typically (though not always) share parts of their behaviors.

To avoid duplication of code, want to write the implementations of these behaviors in just one place.

⇒ inheritance

Inheritance

Basic mechanism of inheritance: *classes*

A class is a data structure that can be

- ▶ *instantiated* to create new objects (“instances”)
- ▶ *refined* to create new classes (“subclasses”)

N.b.: some OO languages offer an alternative mechanism, called *delegation*, which allows new objects to be derived by refining the behavior of existing objects.

Example

```
class A {
    protected int x = 0;
    int m() { x = x+1; return x; }
    int n() { x = x-1; return x; }
}

class B extends A {
    int o() { x = x*10; return x; }
}
```

An instance of `B` has methods `m`, `n`, and `o`. The first two are inherited from `A`.

Late binding

Most OO languages offer an extension of the basic mechanism of classes and inheritance called *late binding* or *open recursion*.

Late binding allows a method within a class to call another method via a special “pseudo-variable” `this`. If the second method is overridden by some subclass, then the behavior of the first method automatically changes as well.

Though quite useful in many situations, late binding is rather tricky, both to define (as we will see) and to use appropriately. For this reason, it is sometimes deprecated in practice.

Examples

```
class E {  
    protected int x = 0;  
    int m() { x = x+1; return x; }  
    int n() { x = x-1; return this.m(); }  
}
```

```
class F extends E {  
    int m() { x = x+100; return x; }  
}
```

Quick check:

- ▶ What does `(new E()).n()` return?
- ▶ What does `(new F()).n()` return?

Calling “super”

It is sometimes convenient to “re-use” the functionality of an overridden method.

Java provides a mechanism called `super` for this purpose.

Example

```
class E {
    protected int x = 0;
    int m() { x = x+1; return x; }
    int n() { x = x-1; return this.m(); }
}

class G extends E {
    int m() { x = x+100; return super.m(); }
}
```

What does `(new G()).n()` return?

Getting down to details
(in the lambda-calculus)...

Simple objects with encapsulated state

```
class Counter {
    protected int x = 1;           // Hidden state
    int get() { return x; }
    void inc() { x++; }
}

void inc3(Counter c) {
    c.inc(); c.inc(); c.inc();
}

Counter c = new Counter();
inc3(c);
inc3(c);
c.get();
```

How do we encode objects in the lambda-calculus?

Objects

```
c = let x = ref 1 in
      {get = λ_:Unit. !x,
       inc = λ_:Unit. x:=succ(!x)};
```

$\implies c : \text{Counter}$

where

```
Counter = {get:Unit→Nat, inc:Unit→Unit}
```

Objects

```
inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);  
⇒ inc3 : Counter → Unit
```

```
(inc3 c; inc3 c; c.get unit);  
⇒ 7
```

Object Generators

```
newCounter =  
  λ_:Unit. let x = ref 1 in  
    {get = λ_:Unit. !x,  
      inc = λ_:Unit. x:=succ(!x)};  
⇒ newCounter : Unit → Counter
```

Grouping Instance Variables

Rather than a single reference cell, the states of most objects consist of a number of *instance variables* or *fields*.

It will be convenient (later) to group these into a single record.

```
newCounter =  
  λ_:Unit. let r = {x=ref 1} in  
    {get = λ_:Unit. !(r.x),  
      inc = λ_:Unit. r.x:=succ(!(r.x))};
```

The local variable `r` has type `CounterRep = {x: Ref Nat}`

Subtyping and Inheritance

```
class Counter {  
    protected int x = 1;  
    int get() { return x; }  
    void inc() { x++; }  
}
```

```
class ResetCounter extends Counter {  
    void reset() { x = 1; }  
}
```

```
ResetCounter rc = new ResetCounter();  
inc3(rc);  
rc.reset();  
inc3(rc);  
rc.get();
```


Subtyping

```
ResetCounter = {get:Unit→Nat,  
                inc:Unit→Unit,  
                reset:Unit→Unit};
```

```
newResetCounter =  
  λ_:Unit. let r = {x = ref 1} in  
    {get    = λ_:Unit. !(r.x),  
     inc    = λ_:Unit. r.x:=succ(!(r.x)),  
     reset  = λ_:Unit. r.x:=1};
```

⇒ newResetCounter : Unit → ResetCounter

Subtyping

```
rc = newResetCounter unit;  
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);  
⇒ 4
```

Simple Classes

The definitions of `newCounter` and `newResetCounter` are identical except for the `reset` method.

This violates a basic principle of software engineering:

Each piece of behavior should be implemented in just one place in the code.

Reusing Methods

Idea: could we just re-use the methods of some existing object to build a new object?

```
resetCounterFromCounter =  
  λc:Counter. let r = {x = ref 1} in  
    {get    = c.get,  
      inc   = c.inc,  
      reset = λ_:Unit. r.x:=1};
```

Reusing Methods

Idea: could we just re-use the methods of some existing object to build a new object?

```
resetCounterFromCounter =  
  λc:Counter. let r = {x = ref 1} in  
    {get    = c.get,  
     inc    = c.inc,  
     reset = λ_:Unit. r.x:=1};
```

No: This doesn't work properly because the `reset` method does not have access to the local variable `r` of the original counter.

⇒ classes

Classes

A class is a run-time data structure that can be

1. *instantiated* to yield new objects
2. *extended* to yield new classes

Classes

To avoid the problem we observed before, what we need to do is to separate the definition of the methods

```
counterClass =  
  λr:CounterRep.  
    {get = λ_:Unit. !(r.x),  
      inc = λ_:Unit. r.x:=succ(!(r.x))};  
⇒ counterClass : CounterRep → Counter
```

from the act of binding these methods to a particular set of instance variables:

```
newCounter =  
  λ_:Unit. let r = {x=ref 1} in  
            counterClass r;  
⇒ newCounter : Unit → Counter
```

Defining a Subclass

```
resetCounterClass =  
  λr:CounterRep.  
    let super = counterClass r in  
      {get    = super.get,  
       inc    = super.inc,  
       reset = λ_:Unit. r.x:=1};  
⇒ resetCounterClass : CounterRep → ResetCounter
```

```
newResetCounter =  
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;  
⇒ newResetCounter : Unit → ResetCounter
```


Overriding and adding instance variables

```
class Counter {  
    protected int x = 1;  
    int get() { return x; }  
    void inc() { x++; }  
}
```

```
class ResetCounter extends Counter {  
    void reset() { x = 1; }  
}
```

```
class BackupCounter extends ResetCounter {  
    protected int b = 1;  
    void backup() { b = x; }  
    void reset() { x = b; }  
}
```

Adding instance variables

In general, when we define a subclass we will want to add new instance variables to its representation.

```
BackupCounter = {get:Unit→Nat, inc:Unit→Unit,  
                 reset:Unit→Unit, backup: Unit→Unit};  
BackupCounterRep = {x: Ref Nat, b: Ref Nat};
```

```
backupCounterClass =  
  λr:BackupCounterRep.  
    let super = resetCounterClass r in  
      {get    = super.get,  
       inc   = super.inc,  
       reset = λ_:Unit. r.x:=!(r.b),  
       backup = λ_:Unit. r.b:=!(r.x)};
```

⇒

```
backupCounterClass : BackupCounterRep → BackupCounter
```

Notes:

- ▶ `backupCounterClass` both extends (with `backup`) and overrides (with a new `reset`) the definition of `counterClass`
- ▶ subtyping is essential here (in the definition of `super`)

```
backupCounterClass =  
  λr:BackupCounterRep.  
    let super = resetCounterClass r in  
      {get      = super.get,  
       inc     = super.inc,  
       reset   = λ_:Unit. r.x:=!(r.b),  
       backup  = λ_:Unit. r.b:=!(r.x)};
```

Calling super

Suppose (for the sake of the example) that we wanted every call to `inc` to first back up the current state. We can avoid copying the code for `backup` by making `inc` use the `backup` and `inc` methods from `super`.

```
funnyBackupCounterClass =  
  λr:BackupCounterRep.  
    let super = backupCounterClass r in  
      {get = super.get,  
       inc = λ_:Unit. (super.backup unit; super.inc unit),  
       reset = super.reset,  
       backup = super.backup};
```

⇒

```
funnyBackupCounterClass : BackupCounterRep → BackupCounter
```