CIS 500

Software Foundations

Fall 2004

29 November

Plans

- This week: Chapter 18/19
- Next week: Chapter 19/Review
- Final exam: Monday, December 20th
- Course status: add together 2 midterm grades

129-160	A	(24%)
90-128	B	(49%)
69-89	C	(20%)
<69	D/F	(7%)

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:
   (a) A translation into a lower-level language
   (b) 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

- objects
- dynamic dispatch
- encapsulation of state
- inheritance
- self (this) and super
- late binding

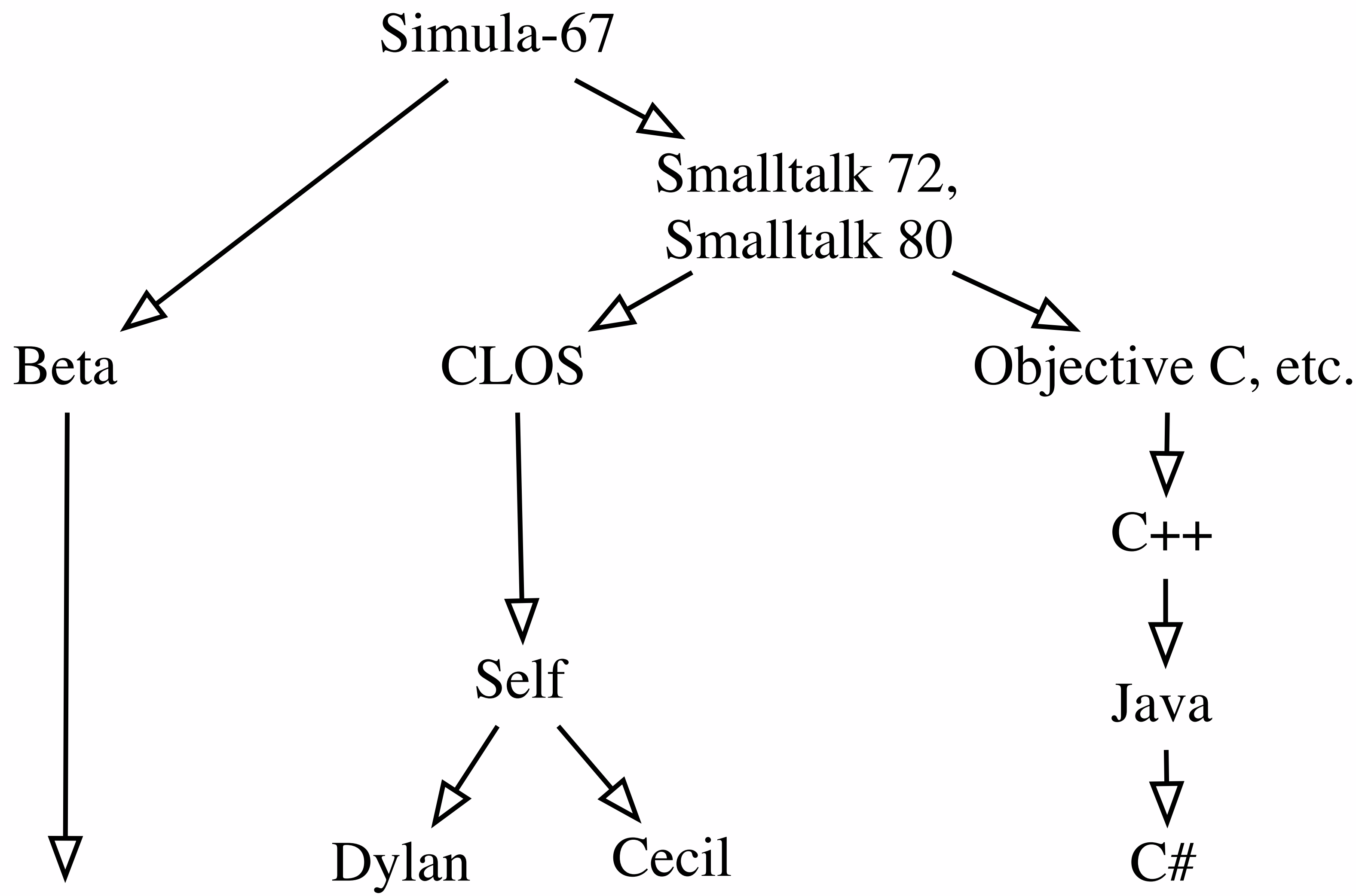
can be understood as "derived forms" in a lower-level language with a rich collection of primitive features:

- (higher-order) functions
- records
- references
- recursion
- subtyping

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

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

History


Simula-67
Smalltalk 72,
Smalltalk 80
Beta
CLOS
Objective C, etc.
C++
Self
Java
Dylan
Cecil
C#

Concepts

The Essence of Objects

What “is” object-oriented programming?

The Essence of Objects

What “is” object-oriented programming?

This question has been a subject of debate for decades. Such arguments are always inconclusive and seldom very interesting.

The Essence of Objects

What “is” object-oriented programming?

This question has been a subject of debate for decades. Such arguments are always inconclusive and seldom very 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 once and for all (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

```
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: `(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 invisible / inaccessible from outside the object

Example

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; }
}
```

Side note: encapsulation

Encapsulation is arguably a little less fundamental than dynamic dispatch, in the sense that there are several OO languages (e.g., CLOS, Dylan, and Cecil) that do not encapsulate state with methods.

These languages are based, instead, on multi-methods, a form of ad-hoc polymorphism.

Although their basic mechanisms are quite different, the higher-level programming idioms (classes, inheritance, etc.) arising in multi-method languages are surprisingly similar to those in "mainstream" OO languages.

(Side note for Java experts: we're also eliding some subtleties involving accessing the `protected` fields of other objects of the same class...)

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.

N.b. 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

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 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.

$\Longrightarrow$ 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 (but fundamentally fairly similar) 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” `self`. 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 tastefully. 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; }
}
```

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)...

Objects

```
 class Counter {
    protected int x = 1;
    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)};
```

⟹ `c : 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
```

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

⟹ newResetCounter : Unit → ResetCounter
```

Subtyping

```
rc = newResetCounter unit;
```

```
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
```

$\Longrightarrow$ `4`

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.

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


CounterRep = {x: Ref Nat};
```

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 x = ref 1 in
              {get   = c.get,
               inc   = c.inc,
               reset = λ_:Unit. 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 x = ref 1 in
              {get   = c.get,
               inc   = c.inc,
               reset = λ_:Unit. x:=1};
```

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

$\Longrightarrow$ 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
```

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 instances 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
```

Calling between methods

Let's define a class of counters with `set`, `get`, and `inc` methods:

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
```

```
setCounterClass =
  λr:CounterRep.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. r.x:=(succ r.x) });
```

Calling between methods

Let's define a class of counters with `set`, `get`, and `inc` methods:

```
SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};
```

```
setCounterClass =
  λr:CounterRep.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. r.x:=(succ r.x) });
```

Bad style: The functionality of `inc` could be expressed in terms of the functionality of `get` and `set`.

Can we rewrite this class so that the `get`/`set` functionality appears just once?

Better...

```
setCounterClass =
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit))});
```

Check: the type of the inner λ-abstraction is `SetCounter→SetCounter`, so the type of the `fix` expression is `SetCounter`.

This is just a definition of a set (record) of mutually recursive functions. (We saw something similar in the `iseven`/`isodd` example in 11.11.)

Note that the fixed point in `setCounterClass =`

```
λr:CounterRep.
  fix
    (λself: SetCounter.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. self.set (succ (self.get unit))});
```

is “closed” — we “tie the knot” when we build the record.

So this does not model the behavior of `self` (or `this`) in real OO languages.

Idea: move the application of `fix` from the class definition...

```
setCounterClass =
  λr:CounterRep.
     λself: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. self.set (succ(self.get unit))};
```

...to the object creation function:

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r);
```

In essence, we are switching the order of `fix` and `λr:CounterRep...`

Note that we have changed the types of classes from... `setCounterClass =`

```
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat.  r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit))});
⟹ setCounterClass : CounterRep → SetCounter
```

... to:

```
setCounterClass =
  λr:CounterRep.
     λself: SetCounter.
       {get = λ_:Unit. !(r.x),
        set = λi:Nat.  r.x:=i,
        inc = λ_:Unit. self.set (succ(self.get unit))};
⟹ setCounterClass : CounterRep → SetCounter → SetCounter
```

Using `self`

Let's continue the example by defining a new class of counter objects (a subclass of set-counters) that keeps a record of the number of times the `set` method has ever been called.

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit,
                inc:Unit→Unit, accesses:Unit→Nat};
```

```
InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

```
instrCounterClass =
  λr:InstrCounterRep.
     λself: InstrCounter.
       let super = setCounterClass r self in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};
```

$\Longrightarrow$

`instrCounterClass : InstrCounterRep → InstrCounter → InstrCounter`

Notes:

- the methods use both `self` (which is passed as a parameter) and `super` (which is constructed using `self` and the instance variables)
- the `inc` in `super` will call the `set` defined here, which calls the superclass `set`
- suptyping plays a crucial role (twice) in the call to `setCounterClass`

One more refinement...

A small fly in the ointment

The implementation we have given for instrumented counters is not very useful because calling the object creation function

```
newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
               fix (instrCounterClass r);
```

will cause the evaluator to diverge!

Intuitively (see TAPL for details), the problem is the “unprotected” use of `self` in the call to `setCounterClass` in `instrCounterClass`:

```
instrCounterClass =
  λr:InstrCounterRep.
     λself: InstrCounter.
       let super = setCounterClass r self in
         ...
```

To see why this diverges, consider a simpler example:

```
ff = λf:Nat→Nat.
       let f′ = f in
       λn:Nat. 0
```

$\Longrightarrow$ `ff : (Nat→Nat) → (Nat→Nat)`

Now:

```
fix ff  ⟶  ff (fix ff)
        ⟶  let f′ = (fix ff) in λn:Nat. 0
        ⟶  let f′ = ff (fix ff) in λn:Nat. 0
        ⟶  uh oh...
```

One possible solution

Idea: "delay" `self` by putting a dummy abstraction in front of it...

```
setCounterClass =
  λr:CounterRep.
  λself: Unit→SetCounter.
    λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat.  r.x:=i,
       inc = λ_:Unit. (self unit).set(succ((self unit).get unit))};
```

$\Longrightarrow$

```
setCounterClass : CounterRep → (Unit→SetCounter) → (Unit→SetCounter)
```

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
              fix (setCounterClass r) unit;
```

Similarly:

```
instrCounterClass =
  λr:InstrCounterRep.
  λself: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r self unit in
         {get = super.get,
          set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
          inc = super.inc,
          accesses = λ_:Unit. !(r.a)};


newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
               fix (instrCounterClass r) unit;
```

Success

This works, in the sense that we can now instantiate `instrCounterClass` (without diverging!), and its instances behave in the way we intended.

Success (?)

This works, in the sense that we can now instantiate `instrCounterClass` (without diverging!), and its instances behave in the way we intended.

However, all the "delaying" we added has an unfortunate side effect: instead of computing the "method table" just once, when an object is created, we will now re-compute it every time we invoke a method!

Section 18.12 in TAPL shows how this can be repaired by using references instead of `fix` to "tie the knot" in the method table.

Recap

Multiple representations

All the objects we have built in this series of examples have type `Counter`.

But their internal representations vary widely.

Encapsulation

An object is a record of functions, which maintain common internal state via a shared reference to a record of mutable instance variables.

This state is inaccessible outside of the object because there is no way to name it. (Instance variables can only be named from inside the methods.)

Subtyping

Subtyping between object types is just ordinary subtyping between types of records of functions.

Functions like `inc3` that expect `Counter` objects as parameters can (safely) be called with objects belonging to any subtype of `Counter`.

Inheritance

Classes are data structures that can be both extended and instantiated.

We modeled inheritance by copying implementations of methods from superclasses to subclasses.

Each class

- waits to be told a record `r` of instance variables and an object `self` (which should have the same interface and be based on the same record of instance variables)
- uses `r` and `self` to instantiate its superclass
- constructs a record of method implementations, copying some directly from `super` and implementing others in terms of `self` and `super`.

The `self` parameter is "resolved" at object creation time using `fix`.

Additional exercise

Take all the examples from this lecture (and the previous one), and recode them in Java.

[Not to be handed in — just for you to check your understanding.]