Object types

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

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

BackupCounter = \{get: Unit→Nat, inc: Unit→Unit, reset: Unit→Unit, backup: Unit→Unit\};

Simple Classes

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

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

where...
CounterRep = \{x: Ref Nat\};
**Subclasses**

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

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

---

**Calling super**

```haskell
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
```

---

**Going Further**
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 =
  Ar:CounterRep.
    \{get = λ_.Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_.Unit. r.x:=(succ r.x)\};
```

Better...

```
setCounterClass =
  Ar:CounterRep.
    fix
      (Aself: 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 =

```
  Ar:CounterRep.
    fix
      (Aself: 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...

```ml
setCounterClass =
  \x: Unit. let r = \x=ref 1 in
  fix (setCounterClass r);
```

...to the object creation function:

```ml
newSetCounter =
  \_: Unit. let r = \x=ref 1 in
  fix (setCounterClass r);
```

In essence, we are switching the order of fix and \texttt{Ar:CounterRep}...

Note that we have changed the types of classes from...

```ml
setCounterClass =
  \x: CounterRep.
    fix
      (\x: SetCounter.
        \{get = \x: Unit. !r.x,
         set = \x: Nat. r.x=i,
         inc = \x: Unit. self.set (succ(self.get unit))\};
      setCounterClass : CounterRep \rightarrow SetCounter
```

... to:

```ml
setCounterClass =
  \x: CounterRep.
    fix
      (\x: SetCounter.
        \{get = \x: Unit. !r.x,
         set = \x: Nat. r.x=i,
         inc = \x: Unit. self.set (succ(self.get unit))\};
      setCounterClass : CounterRep \rightarrow SetCounter \rightarrow 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.

```ml
InstrCounterClass =
  \x: Unit \rightarrow Nat, \textit{set}: \textit{Nat} \rightarrow Unit,
  \textit{ic}: \textit{Unit} \rightarrow Unit, \textit{accesses}: \textit{Unit} \rightarrow \textit{Nat};
```

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

Notes:

- the methods use both \textit{self} (which is passed as a parameter) and \textit{super} (which is constructed using \textit{self} and the instance variables)
- the inc in super will call the set defined here, which calls the superclass set
- supertyping plays a crucial role (twice) in the call to setCounterClass
A small fly in the ointment

The implementation we have given for instrumented counters is not very useful because calling the object creation function
\[
\text{newInstrCounter} = \lambda \cdot \text{Unit. let } r = \{x = \text{ref } 1, a = \text{ref } 0\} \text{ in } \text{fix} (\text{instrCounterClass } r) \text{;}
\]
will cause the evaluator to diverge!

Intuitively (see TAPL for details), the problem is the “unprotected” use of \textit{self} in the call to \textit{setCounterClass} in \textit{instrCounterClass}:
\[
\text{instrCounterClass} = \lambda \cdot \text{InstrCounterRep.}
\]
\[
\text{let self: InstrCounter.}
\]
\[
\text{let super = setCounterClass r self in }
\]

To see why this diverges, consider a simpler example:

\[
\text{ff} = \lambda f : \text{Nat} \to \text{Nat.}
\]
\[
\text{let } f' = f \text{ in }
\]
\[
\lambda n : \text{Nat. } 0
\]
\[
\Rightarrow \text{ff : } (\text{Nat} \to \text{Nat}) \to (\text{Nat} \to \text{Nat})
\]

Now:

\[
\text{fix ff} \rightarrow \text{ff (fix ff)}
\]
\[
\rightarrow \text{let } f' = (\text{fix ff}) \text{ in } \lambda n : \text{Nat. } 0
\]
\[
\rightarrow \text{let } f' = \text{ff (fix ff)} \text{ in } \lambda n : \text{Nat. } 0
\]
\[
\rightarrow \text{uh oh...}
\]

One possible solution

Idea: “delay” \textit{self} by putting a dummy abstraction in front of it...

\[
\text{setCounterClass} = \lambda \cdot \text{CounterRep.}
\]
\[
\text{let self: Unit \to SetCounter.}
\]
\[
\lambda _i : \text{Unit.}
\]
\[
\{\text{get } = \lambda _i : \text{Unit. } !(r.x),
\text{set } = \lambda i : \text{Nat. } \text{r.x} = i,
\text{inc } = \lambda _i : \text{Unit. } (\text{self unit}).\text{set}(\text{succ}((\text{self unit}).\text{get unit}))\};
\]

\[
\Rightarrow
\text{setCounterClass : CounterRep \to (Unit \to SetCounter) \to (Unit \to SetCounter)}
\]

\[
\text{newSetCounter} = \lambda _i : \text{Unit. let } r = \{x = \text{ref } 1\} \text{ in }
\]
\[
\text{fix } (\text{setCounterClass } r) \text{ unit;}
\]
Similarly:

```haskell
instrCounterClass = 
  \x:InstrCounterRep.
  A x: Unit -> InstrCounter.
  A x: Unit.
      let super = setCounterClass r self unit in
        \x: Nat. (\a: Nat. (r.a:= succ(!r.a)); super.set i),
        inc = super.inc,
        accesses = A x: Unit. (!r.a);

newInstrCounter = 
  A x: 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.

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`.