State, Object-Oriented Programming

Explicit State, Polymorphism (CTM 6.1-6.4.4)
Objects, Classes, and Inheritance (CTM 7.1-7.2)

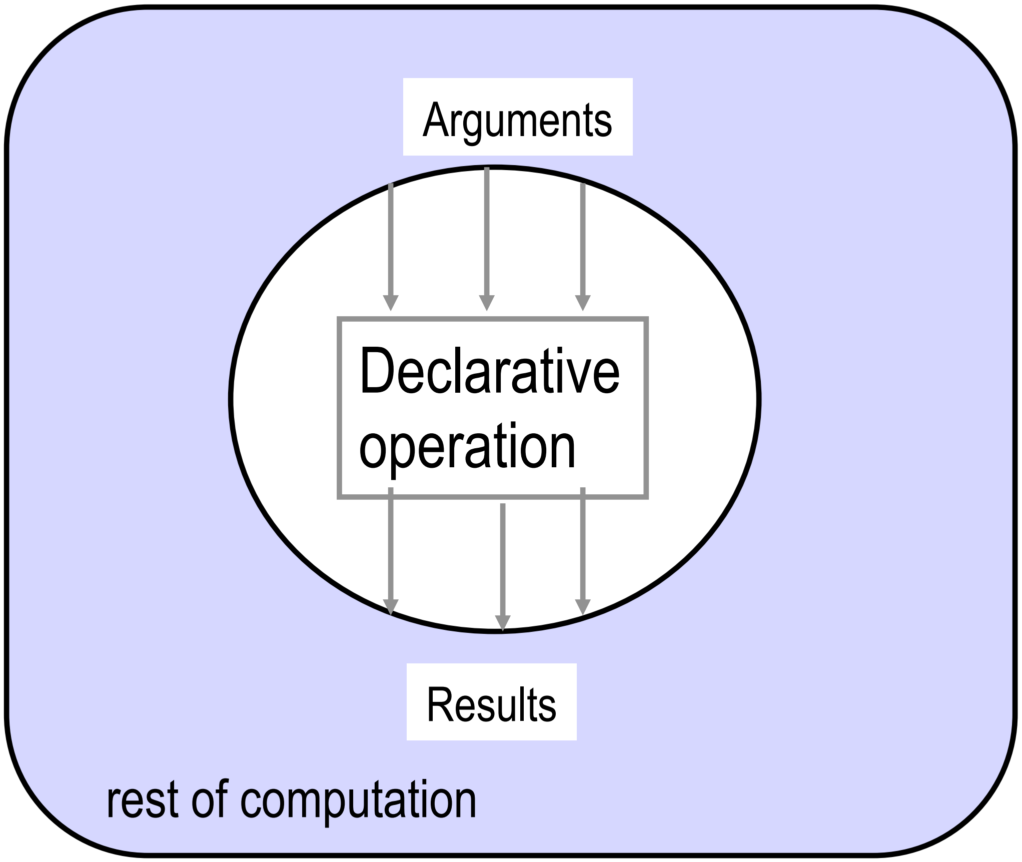
Carlos Varela
Rensselaer Polytechnic Institute
October 27, 2015

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Declarative operations (1)

- An operation is *declarative* if whenever it is called with the same arguments, it returns the same results independent of any other computation state
- A declarative operation is:
  - *Independent* (depends only on its arguments, nothing else)
  - *Stateless* (no internal state is remembered between calls)
  - *Deterministic* (call with same operations always give same results)
- Declarative operations can be composed together to yield other declarative components
  - All basic operations of the declarative model are declarative and combining them always gives declarative components

Declarative operations (2)



Why declarative components (1)

- There are two reasons why they are important:
- *(Programming in the large)* A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
  – The complexity (reasoning complexity) of a program composed of declarative components is the *sum* of the complexity of the components
  – In general the reasoning complexity of programs that are composed of nondeclarative components explodes because of the intimate interaction between components
- *(Programming in the small)* Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).
  – Simple algebraic and logical reasoning techniques can be used

Why declarative components (2)

- Since declarative components are mathematical functions, algebraic reasoning is possible i.e. substituting equals for equals
- The declarative model of CTM Chapter 2 guarantees that all programs written are declarative
- Declarative components can be written in models that allow stateful data types, but there is no guarantee

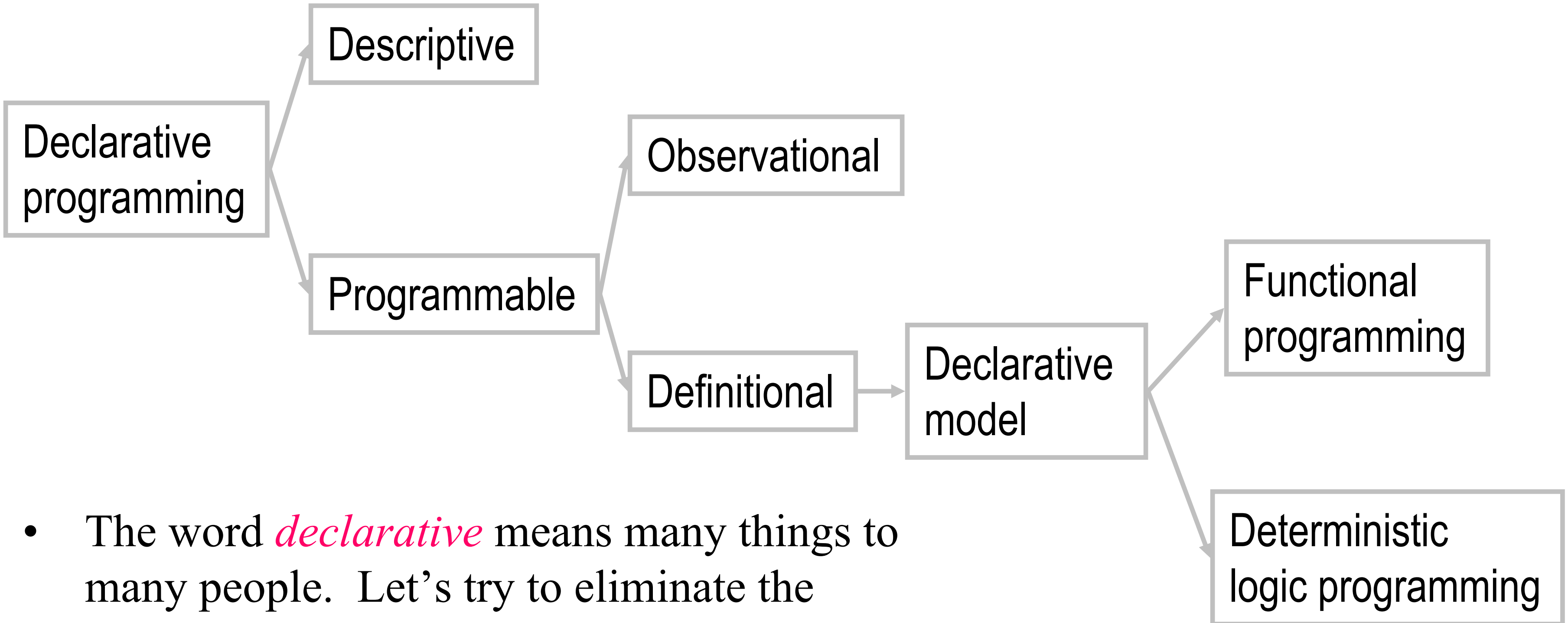
Given

$f(a) = a^2$

We can replace $f(a)$ in any other equation

$b = 7f(a)^2$ becomes $b = 7a^4$

Classification of declarative programming



- The word *declarative* means many things to many people. Let's try to eliminate the confusion.
- The basic intuition is to program by defining the *what* without explaining the *how*

Oz kernel language

The following defines the syntax of a statement, ⟨s⟩ denotes a statement

⟨s⟩ ::=	skip	*empty statement*
|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
|	⟨x⟩ = ⟨v⟩	*variable-value binding*
|	$\langle s_1 \rangle$ $\langle s_2 \rangle$	*sequential composition*
|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
|	proc '{'⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ '}' $\langle s_1 \rangle$ end	*procedure introduction*
|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
|	'{' ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ '}'	*procedure application*
|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*

Why the Oz KL is declarative

- All basic operations are declarative
- Given the components (sub-statements) are declarative,
  - sequential composition
  - local statement
  - procedure definition
  - procedure call
  - if statement
  - case statement

are all declarative (independent, stateless, deterministic).

What is state?

- State is a sequence of values in time that contains the intermediate results of a desired computation
- Declarative programs can also have state according to this definition
- Consider the following program

```
fun {Sum Xs A}
   case Xs
   of X|Xr then {Sum Xr A+X}
   [] nil then A
   end
end

{Browse {Sum [1 2 3 4] 0}}
```

What is implicit state?

The two arguments Xs and A represent an <u>implicit state</u>

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

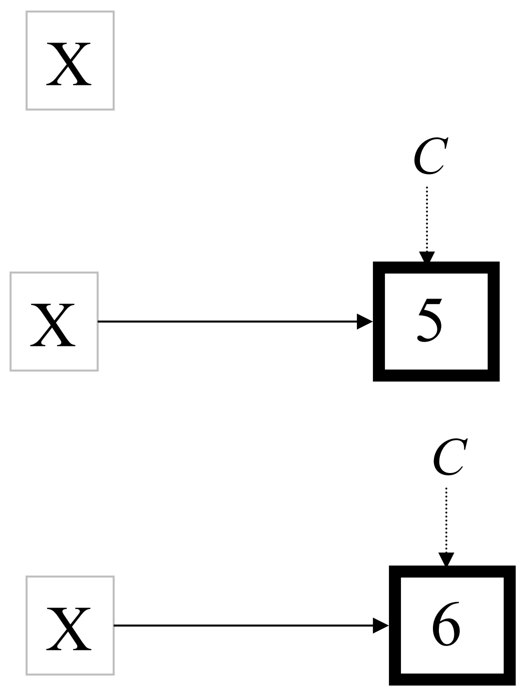
```
fun {Sum Xs A}
   case Xs
   of X|Xr then {Sum Xr A+X}
   [] nil then A
   end
end

{Browse {Sum [1 2 3 4] 0}}
```

What is explicit state: Example?

An unbound variable

A cell *C* is created with initial value 5
X is bound to *C*

The cell C, which X is bound to, is assigned the value 6



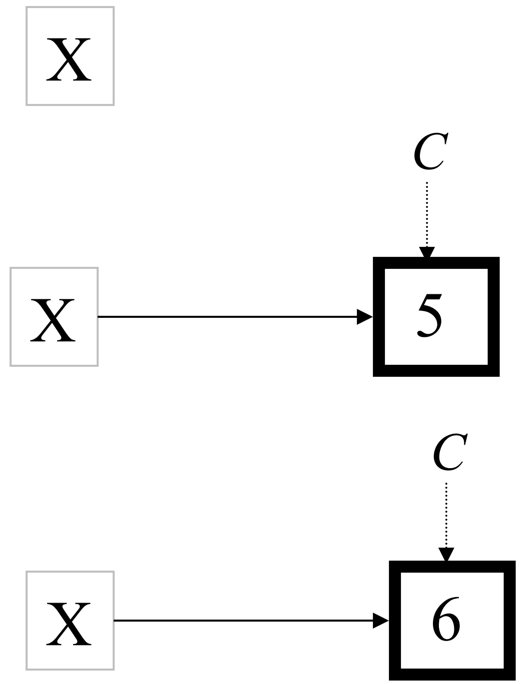
What is explicit state: Example?

An unbound variable

A cell *C* is created with initial value 5

X is bound to *C*

The cell C, which X is bound to, is assigned the value 6



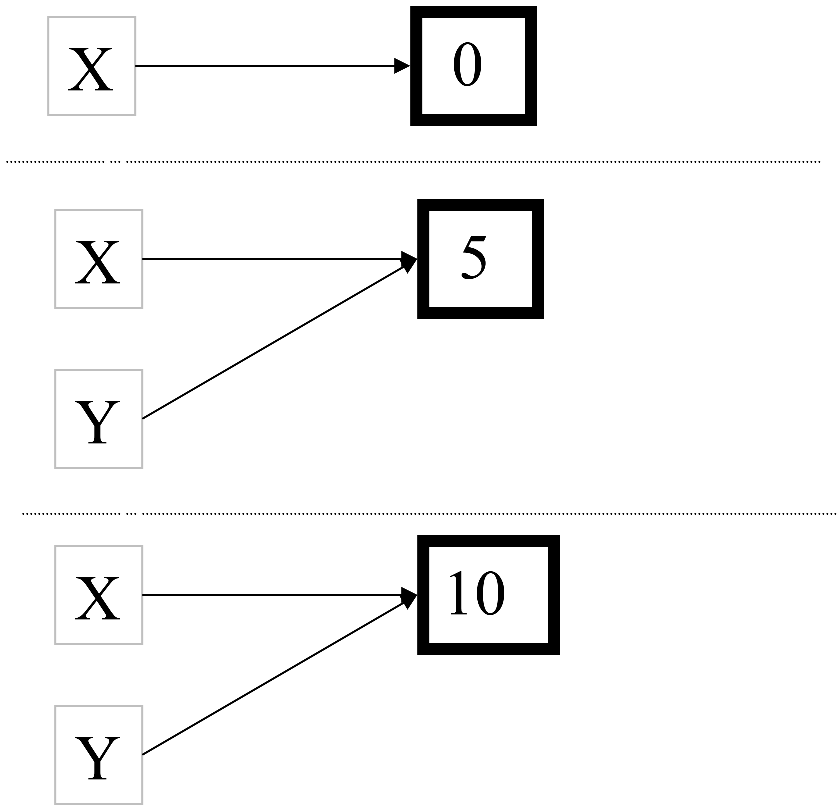
- The cell is a value container with a unique **identity**
- X is really bound to the **identity** of the cell
- When the cell is assigned, X does not change

What is explicit state?

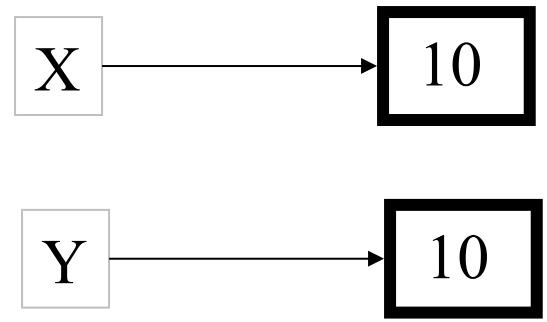
- **X = {NewCell I}**
  - Creates a cell with initial value I
  - Binds X to the identity of the cell
- Example: X = {NewCell 0}
- **{Assign X J}**
  - Assumes X is bound to a cell C (otherwise exception)
  - Changes the content of C to become J
- **Y = {Access X}**
  - Assumes X is bound to a cell C (otherwise exception)
  - Binds Y to the value contained in C

Examples

- X = {NewCell 0}
- {Assign X 5}
- Y = X
- {Assign Y 10}
- {Access X} == 10 % returns true
- X == Y % returns true



Examples



- X = {NewCell 10}
  Y = {NewCell 10}

- X == Y % returns false
- Because X and Y refer to different cells, with different identities

- {Access X} == {Access Y} returns true

The model extended with cells

Semantic stack

w = f(x)
z = person(a:y)
y = $\alpha 1$
u = $\alpha 2$
x

single assignment store

$\alpha 1$: w
$\alpha 2$: x
....
....

mutable store

The stateful model

⟨**s**⟩::=	**skip**	*empty statement*
|	⟨**s**$_1$⟩ ⟨**s**$_2$⟩	*statement sequence*
|	**...**	
|	{NewCell ⟨**x**⟩ ⟨**c**⟩}	***cell creation***
|	{Exchange ⟨**c**⟩ ⟨**x**⟩ ⟨**y**⟩}	***cell exchange***

Exchange: bind ⟨**x**⟩ to the old content of ⟨**c**⟩ and set the content of the cell ⟨**c**⟩ to ⟨**y**⟩

The stateful model

| {NewCell ⟨**x**⟩ ⟨**c**⟩}   ***cell creation***
| {Exchange ⟨**c**⟩ ⟨**x**⟩ ⟨**y**⟩}   ***cell exchange***

Exchange: bind ⟨**x**⟩ to the old content of ⟨**c**⟩ and set the content of the cell ⟨**c**⟩ to ⟨**y**⟩

```
proc {Assign C X} {Exchange C _ X} end
fun {Access C} X in{Exchange C X X}X end
```

C := X is syntactic sugar for **{Assign C X}**
@C is syntactic sugar for **{Access C}**
X=C:=Y is syntactic sugar for **{Exchange C X Y}**

Abstract data types (revisited)

- For a given functionality, there are many ways to package the ADT. We distinguish three axes.
- Open vs. secure ADT: is the internal representation visible to the program or hidden?
- Declarative vs. stateful ADT: does the ADT have encapsulated state or not?
- Bundled vs. unbundled ADT: is the data kept together with the operations or is it separable?
- Let us see what our stack ADT looks like with some of these possibilities

Stack: Open, declarative, and unbundled

- Here is the basic stack, as we saw it before:

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of X|S1 then E=X  S1 end end
fun {IsEmpty S} S==nil end
```

- This is completely unprotected. Where is it useful? Primarily, in small programs in which expressiveness is more important than security.

Stack: *Secure*, declarative, and unbundled

- We can make the declarative stack secure by using a wrapper:

```
local Wrap Unwrap
in
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap nil} end
    fun {Push S E} {Wrap E|{Unwrap S}} end
    fun {Pop S E} case {Unwrap S} of X|S1 then E=X {Wrap S1} end end
    fun {IsEmpty S} {Unwrap S} ==nil end
end
```

- Where is this useful? In large programs where we want to protect the implementation of a declarative component.

Stack: Secure, *stateful*, and unbundled

- Let us combine the wrapper with state:

```
local Wrap Unwrap
in
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap {NewCell nil}} end
    proc {Push W X} C={Unwrap W} in {Assign C X|{Access C}} end
    fun {Pop W} C={Unwrap W} in
        case {Access C} of X|S then {Assign C S} X end
    end
    fun {IsEmpty W} {Access {Unwrap W}}==nil end
end
```

- This version is stateful but lets us store the stack separate from the operations. The same operations work on all stacks.

Stack: Secure, stateful, and *bundled*

- This is the simplest way to make a secure stateful stack:

```
proc {NewStack ?Push ?Pop ?IsEmpty}
        C={NewCell nil}
in
        proc {Push X} {Assign C X|{Access C}} end
        fun {Pop}  case {Access C} of X|S then {Assign C S}  X  end end
        fun {IsEmpty} {Access C} ==nil end
end
```

- Compare the declarative with the stateful versions: the declarative version needs two arguments per operation, the stateful version uses higher-order programming (instantiation)
- With some syntactic support, this is *object-based programming*

Four ways to package a stack

- Open, declarative, and unbundled: the usual declarative style, e.g., in Prolog and Scheme
- Secure, declarative, and unbundled: use wrappers to make the declarative style secure
- Secure, stateful, and unbundled: an interesting variation on the usual object-oriented style
- Secure, stateful, and bundled: the usual object-oriented style, e.g., in Smalltalk and Java
- Other possibilities: there are four more possibilities! **Exercise**: Try to write all of them.

Encapsulated stateful abstract datatypes ADT

- These are stateful entities that can be accessed only by the external interface
- The implementation is not visible outside
- We show two methods to build stateful abstract data types:
  - The functor based approach (record interface)
  - The procedure dispatch approach

The functor-based approach

```
fun {NewCounter I}
   S = {NewCell I}
   proc {Inc} S := @S + 1 end
   proc {Dec} S := @S - 1 end
   fun {Get} @S end
   proc {Put I} S := I end
   proc {Display} {Browse @S} end
in o(inc:Inc dec:Dec get:Get put:Put display:Display)
end
```

The functor-based approach

```
fun {NewCounter I}
   S = {NewCell I}
   proc {Inc} S := @S + 1 end
   proc {Dec} S :=
   fun {Get} @S e
   proc {Put I} S :
   proc {Display} {Browse @S} end
in  o(inc:Inc dec:Dec get:Get put:Put browse:Display)
end
```

The state is collected in cell S
The state is completely encapsulated
i.e. not visible outside

The functor-based approach

```
fun {NewCounter I}
    S = {NewCell I}
    proc {Inc} S := @S + 1 end
    proc {Dec} S := @S - 1 end
    fun {Get} @S end
    proc {Put I} S := I end
    proc {Display} {Browse @S} end
in  o(inc:Inc dec:Dec get:Get put:Put display:Display)
end
```

The interface is created for each instance Counter

The functor-based approach

```
fun {NewCounter I}
   S = {NewCell I}
   proc {Inc} S := @S + 1 end
   proc {Dec} S := @S - 1end
   fun {Get} @S end
   proc {Put I} S := I end
   proc {Display} {Browse S.v} end
in o(inc:Inc dec:Dec get:Get put:Put display:Display)
end
```

functions that access
the state by lexical scope

Call pattern

```
declare C1 C2
C1 = {NewCounter 0}
C2 = {NewCounter 100}

{C1.inc}
{C1.display}

{C2.dec}
{C2.display}
```

Defined as a functor

```
functor Counter
export inc:Inc dec:Dec get:Get put:Put display:Display init:Init
define
    S
    proc {Init init(I)} S = {NewCell I} end
    proc {Inc} S := @S + 1 end
    proc {Dec} S := @S - 1 end
    fun {Get} @S end
    proc {Put I} S := I end
    proc {Display} {Browse @S} end
end
```

Functors

- Functors have been used as a specification of modules
- Also functors have been used as a specification of abstract datatypes
- How to create a stateful entity from a functor?

Explicit creation of objects from functors

- Given a variable F that is bound to a functor
- **[O] = {Module.apply [F]}**
  creates stateful ADT object O that is an instance of F

- Given the functor F is stored on a file ’f.ozf’
- **[O] = {Module.link [’f.ozf’]}**
  creates stateful ADT object O that is an instance of F

Defined as a functor

```
functor Counter
export inc:Inc dec:Dec get:Get put:Put display:Display init:Init
define
    S
    proc {Init init(I)} S = {NewCell I} end
    proc {Inc} S := @S + 1 end
    proc {Dec} S := @S - 1 end
    fun {Get} @S end
    proc {Put I} S := I end
    proc {Display} {Browse @S} end
end
```

Pattern of use

```
fun {New Functor Init}
   M in
   [M] = {Module.apply [Functor]}
   {M.init Init}
   M
End

declare C1 C2
C1 = {New Counter init(0)}
C2 = {New Counter init(100)}
{C1.inc} {C1.put 50} {C1.display}
{C2.dec} {C2.display}
```

Generic function to create objects from functors

Object interface is a record with procedure values inside fields

The procedure-based approach

```
fun {Counter}
   S
   proc {Inc inc(Value)} S := @S + Value end
   proc {Display display} {Browse @S} end
   proc {Init init(I)} S = {NewCell I} end
   D = o(inc:Inc display:Display init:Init)
in proc{$ M} {D.{Label M} M} end
end
```

The procedure-based approach

```
fun {Counter}
   S
   ...
   D = o(inc:Inc display:Display init:Init)
in  proc{$ M}  {D.{Label M} M} end
end
fun {New Class InitialMethod}
   O = {Class}
in {O InitialMethod} O end
```

Example

- The following shows how an object is created from a class using the procedure **`New/3`**, whose first argument is the class, the second is the initial method, and the result is the object.
- **`New/3`** is a generic procedure for creating objects from classes.

```
declare C = {New Counter init(0)}
{C display}
{C inc(1)}
{C display}
```

Object interface is as a procedure of one argument, which expects a record

Object-oriented programming

- Supports
  - Encapsulation
  - Compositionality
  - Instantiation
- Plus
  - Inheritance

Inheritance

- Programs can be built in hierarchical structure from ADT's that depend on other ADT's (Components)
- Object-oriented programming (inheritance) is based on the idea that ADTs have so much in common
- For example, sequences (stacks, lists, queues)
- Object oriented programming enables building ADTs incrementally, through *inheritance*
- An ADT can be defined to *inherit* from another abstract data type, substantially sharing functionality with that abstract data type
- Only the difference between an abstract datatype and its ancestor has to be specified

What is object-oriented programming?

- OOP (Object-oriented programming) = encapsulated state + inheritance
- Object
  - An entity with unique identity that encapsulates state
  - State can be accessed in a controlled way from outside
  - The access is provided by means of methods (procedures that can directly access the internal state)
- Class
  - A specification of objects in an incremental way
  - Incrementality is achieved inheriting from other classes by specifying how its objects (instances) differ from the objects of the inherited classes

Instances (objects)

Interface (what methods are available)
State (attributes)
Procedures (methods)

Classes (simplified syntax)

A class is a statement

```
class ⟨ClassVariable⟩
   attr
      ⟨AttrName1⟩
         :
      ⟨AttrNameN⟩
   meth ⟨Pattern1⟩  ⟨Statement⟩  end
                 :
   meth ⟨PatternN⟩  ⟨Statement⟩  end
end
```

Classes (simplified syntax)

A class can also be a value that can be in an expression position

```
class $
    attr
       ⟨AttrName1⟩
          :
       ⟨AttrNamen⟩
    meth ⟨Pattern⟩ ⟨Statement⟩ end
                  :
    meth ⟨Pattern⟩ ⟨Statement⟩ end
end
```

Classes in Oz

The class Counter has the syntactic form

```
class Counter
      attr val
      meth display
         {Browse @val}
      end
      meth inc(Value)
          val := @val + Value
      end
      meth init(Value)
          val := Value
      end
end
```

Attributes of Classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

val is an attribute:
a modifiable cell
that is accessed by the
atom val

Attributes of classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

the attribute val is accessed by the operator @val

Attributes of classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

the attribute val is assigned by the operator := as val := ...

Methods of classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

methods
are statements
method head is a
record (tuple) pattern

Classes in Oz

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

Example

- An object is created from a class using the procedure **`New/3`**, whose first argument is the class, the second is the initial method, and the result is the object (such as in the functor and procedure approaches)
- **`New/3`** is a generic procedure for creating objects from classes.

```
declare C = {New Counter init(0)}
{C display}
{C inc(1)}
{C display}
```

Summary

- A class *X* is defined by:
  - **class** *X* ... **end**
- Attributes are defined using the attribute-declaration part before the method-declaration part:
  - **attr** A_1 ... A_N
- Then follows the method declarations, each has the form:
  - **meth** *E* *S* **end**
- The expression *E* evaluates to a method head, which is a record whose label is the method name.

Summary

- An attribute *A* is accessed using `@A`.
- An attribute is assigned a value using `A := E`
- A class can be defined as a value:
- `X = class $ ... end`

Attribute Initialization

- Stateful (may be updated by :=)
- Initialized at object creation time, all instances have the initial balance = 0

```
• class Account
      attr balance:0
      meth … end
      …
   end
```

In general the initial value of an attribute could be any legal value (including classes and objects)

Attribute Initialization

- Initialization by instance

```
class Account
    attr balance
    meth init(X) balance := X end
    …
end
```

- O1 = {New Account init(100)}
- O2 = {New Account init(50)}

Attribute Initialization

- Initialization by brand

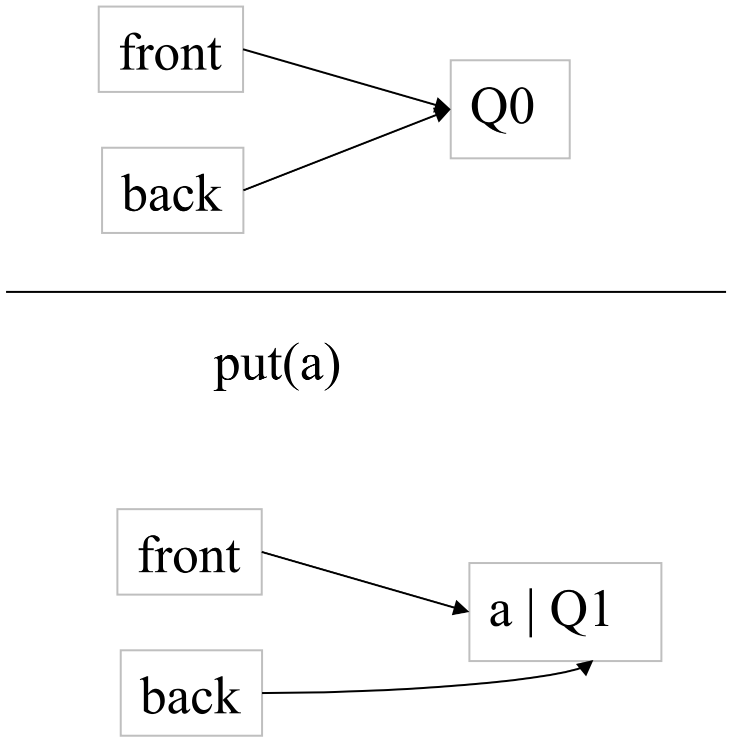
```
declare L=linux
class RedHat
   attr ostype:L
   meth get(X) X = @ostype end
end
class SuSE
   attr ostype:L
   meth get(X) X = @ostype end
end
class Debian
   attr ostype:L
   meth get(X) X = @ostype end
end
```

Example

```
class Queue
  attr front back count
  meth init
    Q in
    front := Q  back := Q  count := 0
  end
  meth put(X)
    Q in
    @back = X|Q
    back := Q
    count := @count + 1
  end
   ...
end
```

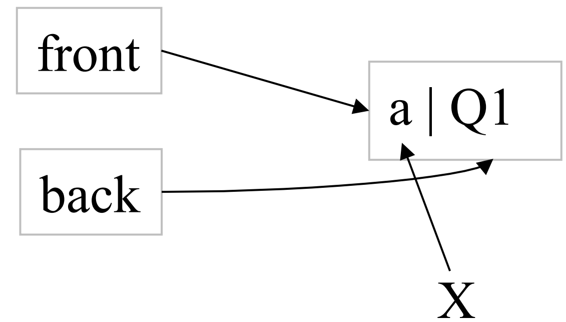
Example

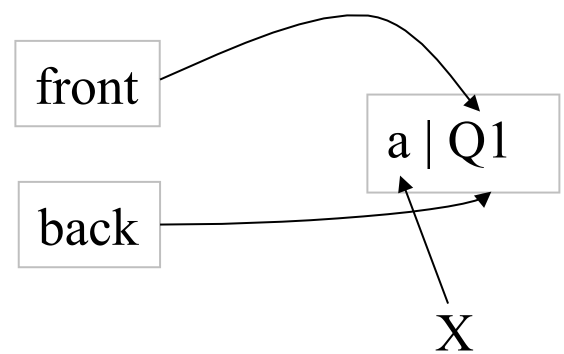
```
class Queue
  attr front back count
  meth init
    Q in
    front := Q  back := Q  count := 0
  end
  meth put(X)
    Q in
    @back = X|Q
    back := Q
    count := @count + 1
  end
   ...
end
```



Example

```
class Queue
   attr front back count
    ...
   meth get(?X)
      Q in
      X|Q = @front
      front := Q
      count := @count - 1
   end
   meth count(?X) X = @count end
   ...
end
```



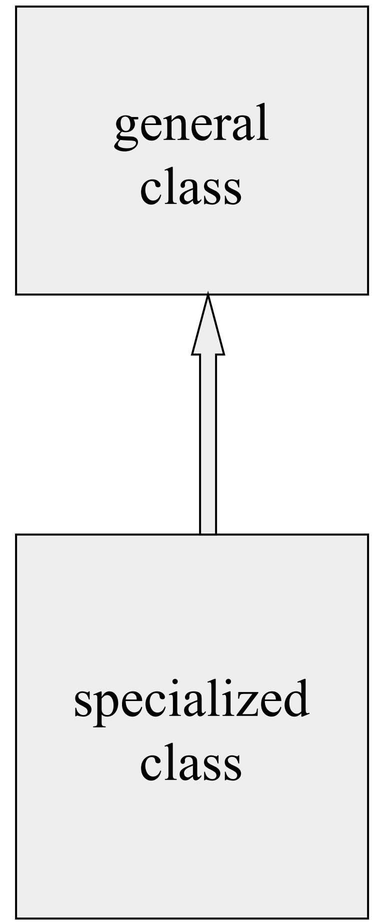


Classes as incremental ADTs

- Object-oriented programming allows us to define a class by extending existing classes
- Three things have to be introduced
  - How to express inheritance, and what does it mean?
  - How to access particular methods in the new class and in preexisting classes
  - Visibility – what part of the program can see the attributes and methods of a class
- The notion of delegation as a substitute for inheritance

Inheritance

- Inheritance should be used as a way to specialize a class *while retaining the relationship between methods*
- In this way it is a just an extension of an ADT
- The other view is inheritance is just a (lazy) way to construct new abstract data types !
- No relationships are preserved



Inheritance

```
class Account
  attr balance:0
  meth transfer(Amount)
    balance := @balance+Amount
  end
  meth getBal(B)
    B = @balance
  end
end

A={New Account transfer(100)}
```

Inheritance II

The class VerboseAccount has the methods:
transfer, getBal, and verboseTransfer

Conservative extension

```
class VerboseAccount
  from Account
  meth verboseTransfer(Amount)
         ...
  end
end
```

Inheritance II

Non-Conservative extension

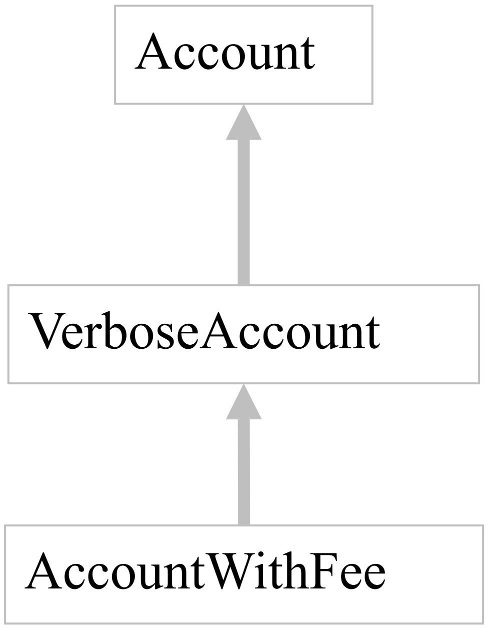
```
class AccountWithFee
  from VerboseAccount
  attr fee:5
  meth transfer(Amount)
    ...
  end
end
```

The class AccountWithFee has the methods:
transfer, getBal, and verboseTransfer
The method transfer has been redefined (overridden) with another definition

Inheritance II

Non-Conservative extension

```
class AccountWithFee
   from VerboseAccount
   attr fee:5
   meth transfer(Amount)
      ...
   end
end
```

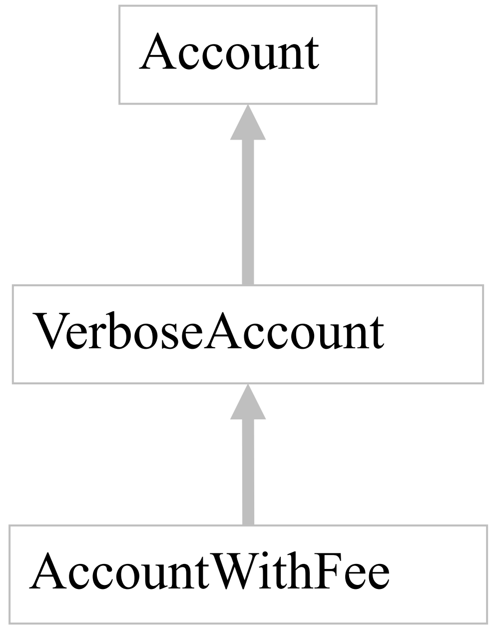


Polymorphism

The ability for operations to take objects (instances) of different types.

For example, the transfer method can be invoked in account object instances of three different classes.

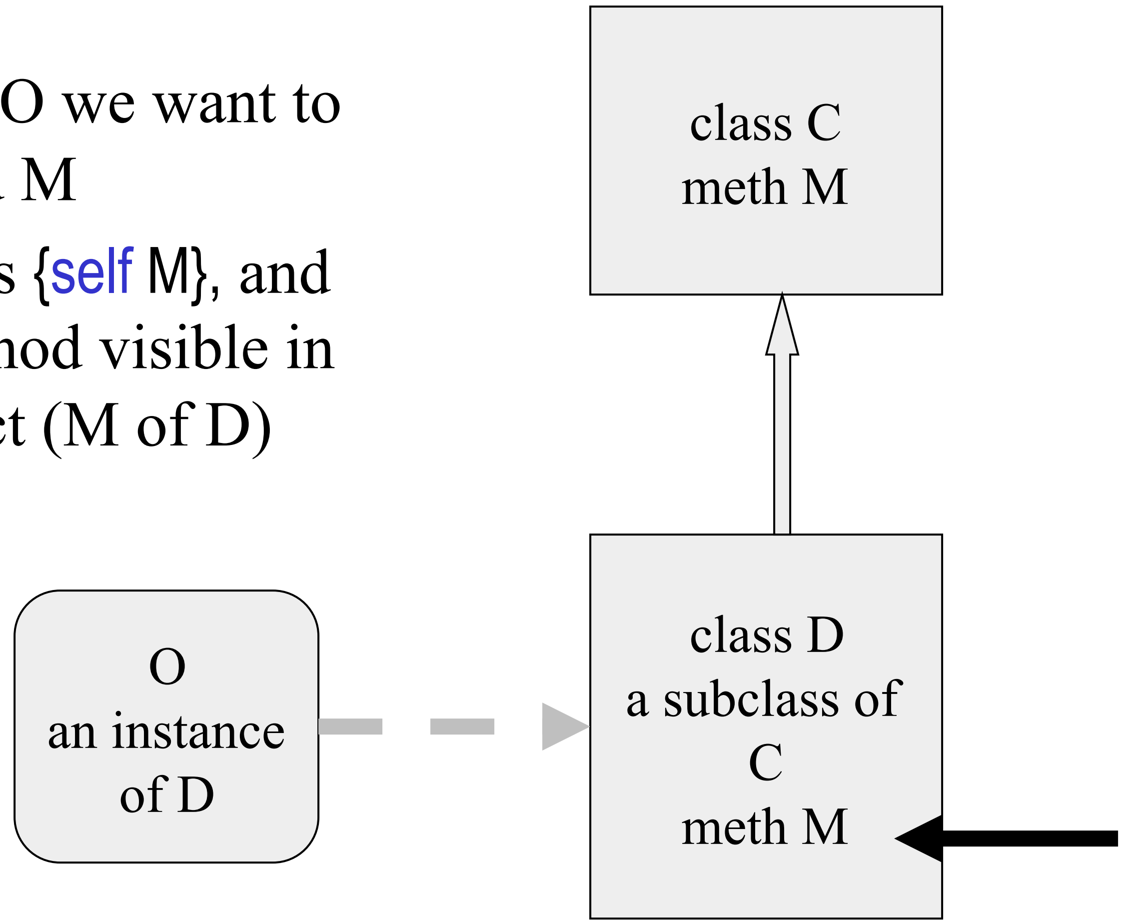
The most specific behavior should be executed.



Static and dynamic binding

Dynamic binding

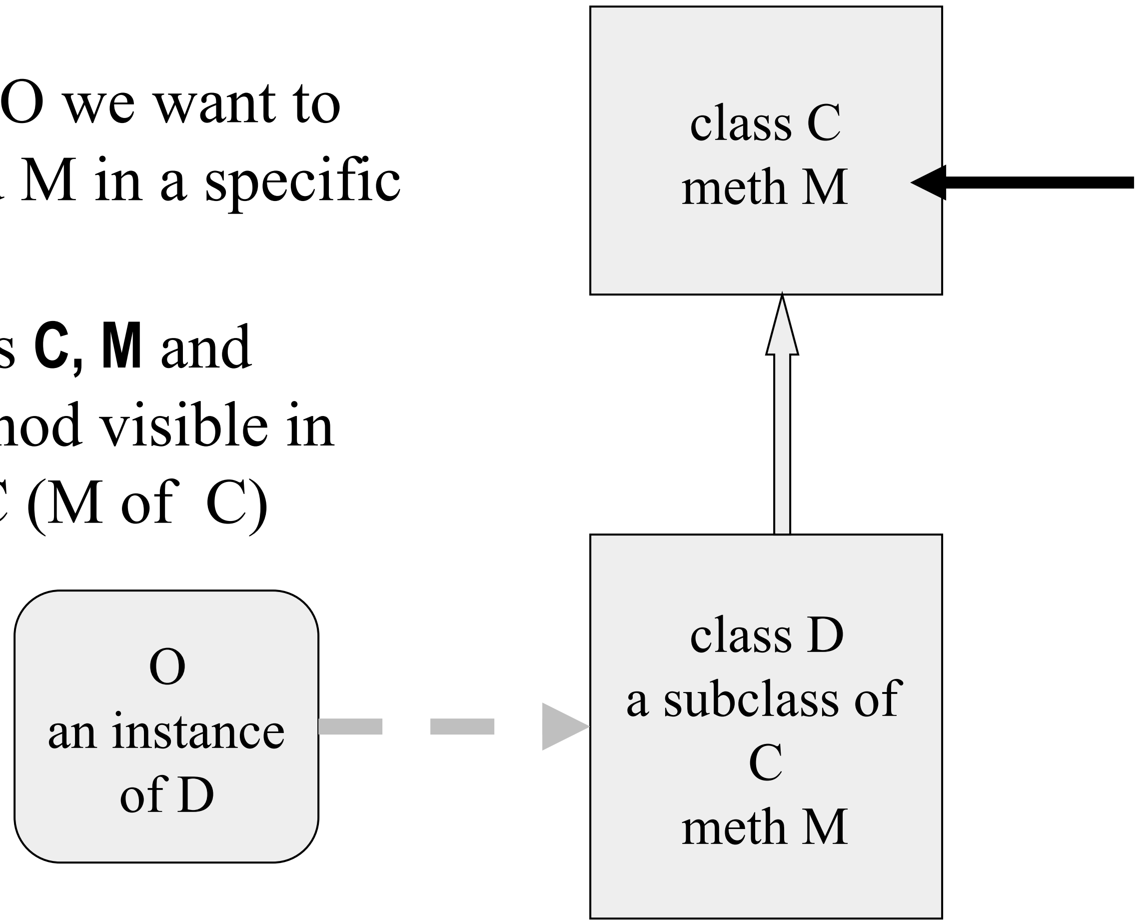
- Inside an object O we want to invoke a method M
- This is written as {self M}, and chooses the method visible in the current object (M of D)



Static and dynamic binding

Static binding

- Inside an object O we want to invoke a method M in a specific (super) class
- This is written as **C, M** and chooses the method visible in the super class C (M of C)



Static method calls

- Given a class and a method head `m(…)`, a static method-call has the following form:

  `C, m(…)`

- Invokes the method defined in the class argument.
- A static method call can only be used inside class definitions.
- The method call takes the current object denoted by **`self`** as implicit argument.
- The method `m` could be defined in the class `C,` or inherited from a super class.

Exercises

63. Do Java and C++ object abstractions completely encapsulate internal state? If so, how? If not, why?
64. Do Java and C++ enable static access to methods defined in classes arbitrarily high in the inheritance hierarchy? If so, how? If not, why?
65. Exercise CTM 7.9.1 (pg 567)
66. Exercise CTM 7.9.7 (pg 568)