A Theory of Objects
(M. Abadi & L. Cardelli)

CS520: Programming Languages
Santiago Pericas-G.
Boston University
Fall 2001
Chapter 4: Object-Based Languages
Objects without Classes

- Object-based languages are intended to be simpler and more flexible.

- They are characterized by the absence of classes.

- An object can be defined directly (i.e. by listing its attributes) or by cloning other objects.
Objects

ObjectType Cell is
  var contents: Integer;
  method get(): Integer;
  method set(n:Integer);
end;

object cell:Cell is
  var contents: Integer := 0;
  method get(): Integer is return self.contents end;
  method set(n:Integer) is self.contents := n end;
end;
Object Creation

- Procedures can be used to generate a uniform collection of objects:

```
procedure newCell(m:Integer):Cell is
  object cell:Cell is
    var contents: Integer := m;
    method get(): Integer is ... end;
    method set(n:Integer) is ... end;
  end; return cell;
end;
var cellInstance:Cell := newCell(0);
```
Prototypes and Clones

- Alternative approach to object creation (so-called prototype-based languages):
  i. generate *stock* objects from *prototype* objects,
  ii. customize *stock* objects.

- Stock objects can be created by shallow-copying prototype objects (a.k.a. cloning).

- Cloning is a bit like `new`, but operates on objects instead of classes.
Cloning and customizing:

\[
\textbf{var} \ \text{cellClone:Cell} = \textbf{clone} \ \text{cellInstance};
\]

\[
\text{cellClone.contents} := 3;
\]

\[
\text{cellClone.get} :=
\]

\[
\textbf{method}(): \text{Integer is}
\]

\[
\textbf{return} \ \text{self.contents} * \text{self.contents};
\]

\[
\textbf{end};
\]
Example: Method Update

object reCell:ReCell is
  var contents: Integer := 0;
  method get(): Integer is return self.contents end;
  method set(n:Integer) is
    let x = self.get();
    self.restore := method() is self.contents := x end;
    self.contents := n;
  end;
  method restore() is self.contents := 0 end;
end;
Inheritance: Embedding vs. Delegation

- Cloning and update are not sufficiently flexible: they always preserve the shape of the prototype.

- One possible solution is to allow for new attributes to be added to an object (a.k.a. object extension).

- An alternative, is to build new objects that include attributes from existing ones + new attributes.
Embedding

Main idea: *host* objects contain copies of attributes that belong to *donor* objects.

It provides the simplest semantics of self as a receiver of an invocation.

Invoking an inherited method is identical to invoking an original method (i.e. no need for a special method invocation mechanism).
Figure 4-1. Embedding.
Embedding

object cell:Cell is
    var contents: Integer := 0;
    method get(): Integer is
        return self.contents;
    end;
    method set(n:Integer) is
        self.contents := n;
    end;
end;
Explicit Embedding

object reCellExp:ReCell is
  var contents: Integer := cell.contents;
  var backup: Integer := 0;
  method get():Integer is return embed cell.get() end;
  method set(n:Integer) is
    self.backup := self.contents;
    embed cell.set(n);
  end;
  method restore() is self.contents := self.backup end;
end;
Implicit Embedding

object reCellImp:ReCell extends cell is
    var backup: Integer := 0;
    override set(n:Integer) is
        self.backup := self.contents;
        embed cell.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
Delegation

- Main idea: **host** objects contain references to **donor** objects.

- Field accesses and method invocations may be re-directed from the host to the donor.

- Method inheritance:
  - Embedding: at the time of method creation.
  - Delegation: at the time of method invocation.
Delegation

Figure 4-2. Delegation.
Explicit Delegation

object reCellExp:ReCell is
  var contents: Integer := cell.contents;
  var backup: Integer := 0;
method get():Integer is return delegate cell.get() end;
method set(n:Integer) is
  self.backup := self.contents;
  delegate cell.set(n);
end;
method restore() is self.contents := self.backup end;
end;
Implicit Delegation

object reCellImp:ReCell extends cell is
    override contents:Integer := cell.contents;
    var backup: Integer := 0;
    override set(n:Integer) is
        self.backup := self.contents;
        delegate cell.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;