Admistrative Announcments

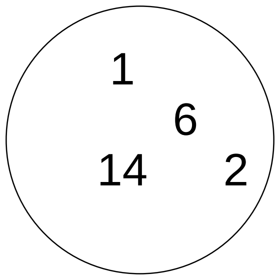
- Extra lecture on Monday, March 4 in 373 Soda. Read Chapter 13, Sections 1-6.
- Lecture on Tuesday, March 5. Read remainder of Chapter 13, focusing on type reconstruction in Section 9.
- Lecture on Thursday, March 7. Read paper on static type checking in OO languages by Palsberg and Schwartzbach.
- There will be no lecture on Tuesday, March 12, but Midterm 1 will be given out that day (to be returned within 24 hours).
- Thursday, March 14 will be a guest lecture on effect systems by Manuel Fahndrich.

Project Ideas

- Comparative evaluation of languages in a particular problem domain, e.g., languages for distributed computing, database queury languages, scripting language.
- Develop semantics for small but unusual languages, e.g., Split-C (subset), Verilog, graphical languages.
- Compare solutions to a particular language design problem, e.g., polymorphism (both implementation issues and usability), exception handling, parallelism, real-time programming.
- Literature search/evaluation on a current topic of programming language research, e.g., abstract interpretation, partial evaluation, formal semantics.
- Study and propose applications for the use of advanced programming ideas in compilation.
- Develop a rigorous correctness proof for a complex algorithm or system (probably concurrent).

Inheritance

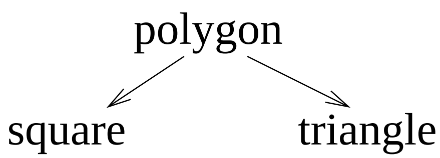
Encapsulation: protecting data from external view/modification



set of integers with operations

create, insert, isin, print

Inheritance: meaning depends on object



Inheritance

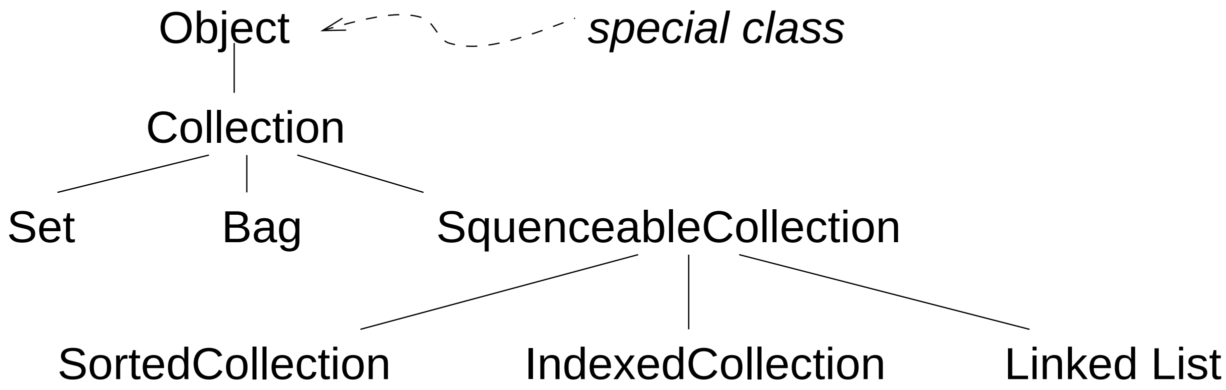
Inheritance is essentially a naming issue – what names are visible in a given scope or by a given object.

Uses of inheritance:

- Interface reuse – document a common interface between a set of related modules. (IS-A)
- Explicitly document subtyping relation between two sets of objects, i.e., that the objects of type A are a subset of the objects of type B. (IS-A)
- Code reuse – share general-purpose state and methods associated with that state. (HAS-A or AS-A)

There are three different problems: sharing state, sharing code, and sharing interfaces. How many language mechanisms do we need?

Smalltalk



Single inheritance hierarchy shown. Subclass may be subset.

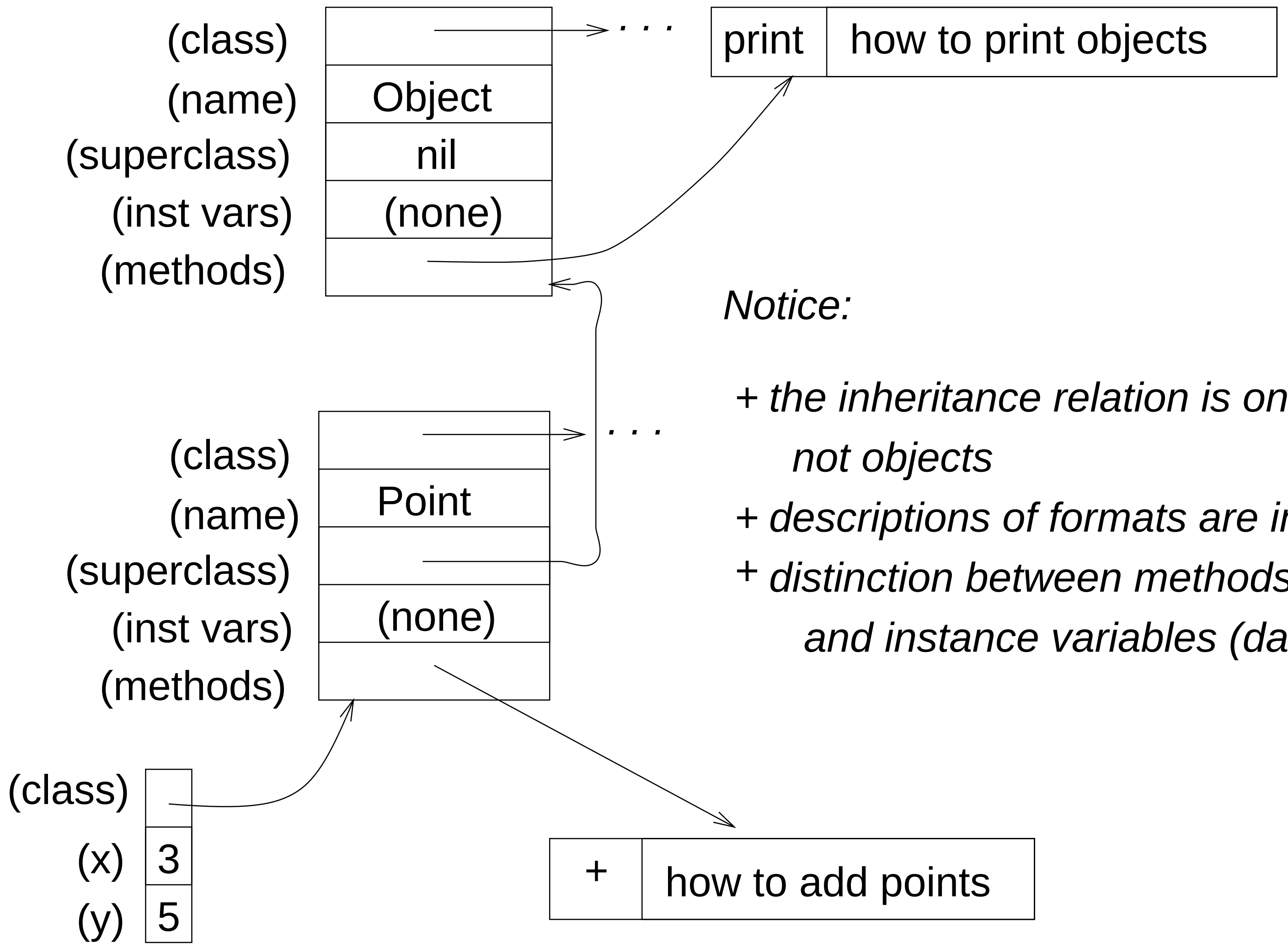
A subclass inherits variables and methods from its superclass.

A subclass may declare new variables (with distinct names from super).

Methods in the subclass take precendence over methods of the same name in the super class. Explicit use of "super" overrides this rule.

A message to "self" is a message to the object (which may be in a subclass) of the current method.

Smalltalk Objects



Notice:

+ *the inheritance relation is on classes, not objects*
+ *descriptions of formats are in the classes*
+ *distinction between methods (operations) and instance variables (data)*

Self

Design goal: exploratory programming (rapid prototyping)

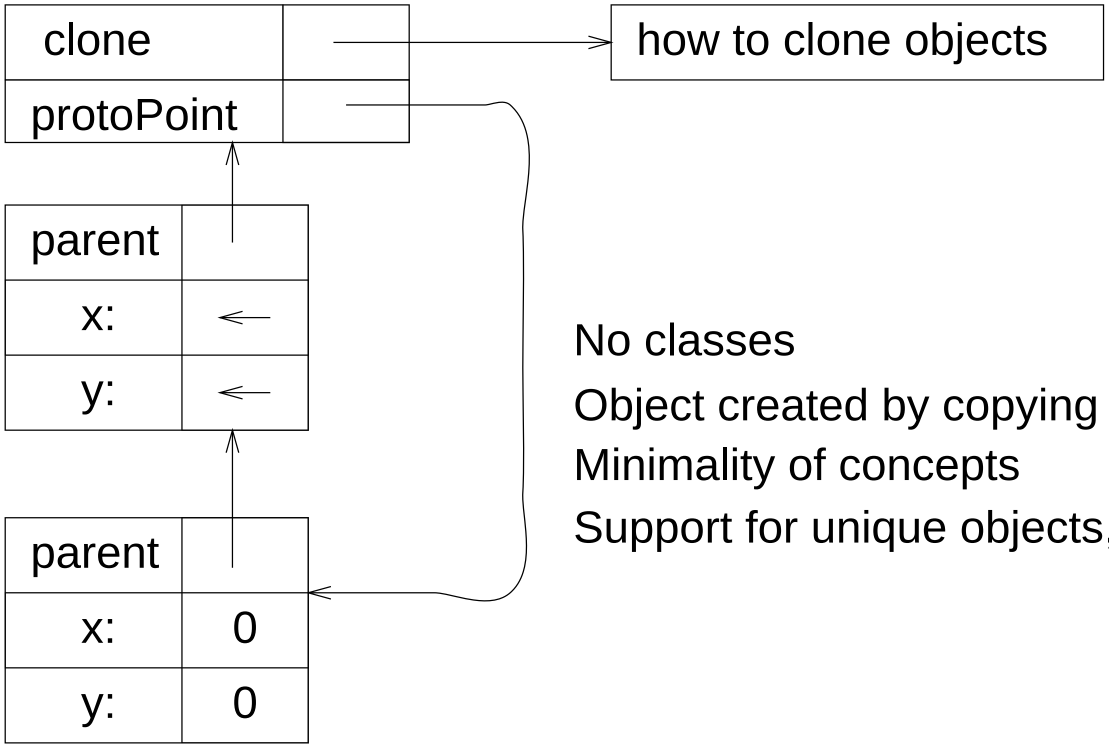
- dynamically typed (no type declarations)
- automatic memory management (gc)

prototyping metaphor: objects are created from other "prototype" objects, not as instances of classes. Neither classe nor variables exist.

Classes vs. Objects

	class-based	**object-based**
inheritance relations	instance of subclass of	inherits from
creation one-of-a-kind	use plan need extra object class	clone no extra object needed
infinite regress	class of ...	non required

Self Objects



No classes

Object created by copying a prototype object

Minimality of concepts

Support for unique objects, i.e., one-of-a-kind.

Syntax for HOOK.

$$\begin{aligned} E &\in \text{Exp} \\ M &\in \text{Message} = \text{Identifer} \\ I &\in \text{Identifer} \\ L &\in \text{Lit} = \text{Intlit} + \text{Boollit} + \ldots \end{aligned}$$

$$\begin{aligned} E \; ::= \; & L \\ \mid \; & I \\ \mid \; & (\texttt{method}\ M_{message}\ (I_{self}\ I_{formal}*)\ E_{body}) \\ \mid \; & (\texttt{object-compose}\ E_{obj1}\ E_{obj2}) \\ \mid \; & (\texttt{null-object}\ ) \\ \mid \; & (\texttt{send}\ M_{message}\ E_{receiver}\ E_{arg}*) \end{aligned}$$

HOOK

- (`method` $M_{message}$ (I_{self} $I_{formal}*$) E_{body})

  Create a new object that responds to message $M_{message}$ by executing the expression E_{body}.

- (`object-compose` E_{obj1} E_{obj2})

  Create an objects that responds to the methods of both objects E_{obj1} and E_{obj2}.

- (`send` $M_{message}$ $E_{receiver}$ $E_{arg}*$)

  Send a method inovocation $M_{message}$ to object $E_{receiver}$ with arguments $E_{arg}*$.

- `null-object`

  Create a null object that responds to no methods.

HOOPLA

$$
\begin{aligned}
P &::= (\texttt{program}\ E_{body}\ D_{def}*) \\
D &::= (I_{name}\ E_{value}) \\
E &::= \ldots\text{[HOOK expressions]} \\
&\quad\mid (\texttt{object}\ E_{object}*) \\
&\quad\mid (\texttt{class}\ (I_{init}*)\ E_{body}) \\
&\quad\mid (\texttt{lambda}\ (I_{formal}*)\ E_{body}) \\
&\quad\mid (E_{rator}\ E_{rand}*) \\
&\quad\mid (\texttt{let}\ ((I_{name}\ E_{value})*)\ E_{body}) \\
&\quad\mid (\texttt{if}\ E_{test}\ E_{consequent}\ E_{alternative}) \\
&\quad\mid (\texttt{begin}\ E_{sequent}*)
\end{aligned}
$$

HOOPLA Example

```
(define point
  (class (init-x init-y)
    (method x (self) init-x)
    (method y (self) init-y)
    (method move (self cx dy)
       (object (send make point
                     (send + (send x self) dx)
                     (send + (send y self) dy))
               self) ;; allows mixins
       )))

(define color
   (class (clr)
     (method color (self) clr)
     (method new-color (self new)
        (object (send make color new)
                self))))
```

HOOPLA Desugaring

We will desugar the non-OO things into objects.

$$\mathcal{D}[\![(\texttt{object }\ )]\!] = (\texttt{null-object }\ )$$
$$\mathcal{D}[\![(\texttt{object }\ E_1\ E_{rest}*)]\!] = (\texttt{object-compose }\ E_1(\texttt{object }\ E_{rest}*))$$

$$\mathcal{D}[\![(\texttt{class }\ (I_{init}*)\ E_{instance}*)]\!] = (\texttt{method }\ \texttt{make }\ (I_{ignore}\ I_{init}*)) \\ (\texttt{object }\ E_{instance}*))$$

$$\mathcal{D}[\![(\texttt{lambda }\ (I_{formal}*)E_{body})]\!] = (\texttt{method }\ \texttt{call}(I_{ignore}\ I_{formal*})E_{body})$$

$$\mathcal{D}[\![(E_{rator}\ E_{rand}*)]\!] = ((\texttt{lambda }\ (I_{var}*)E_{body})E_{val}*)$$

$$\mathcal{D}[\![(letc((I_{var}\ E_{val})*)E_{body})]\!] = ((\texttt{lambda }\ (I_{var}*)\ E_{body})\ E_{val}*)$$

$$\mathcal{D}[\![(\texttt{if }\ E_{test}\ E_{con}\ E_{alt})]\!] = (\texttt{send }\ \texttt{if-true }\ E_{test}\ (\texttt{lambda }\ ()E_{con}) \\ (\texttt{lambda }\ ()E_{alt}))$$

HOOK Translation

$$\mathcal{T}[\![(\texttt{method}\ M_{message}(I_{self}\ I_{formal}*)\ E_{body})]\!] =$$
$$(\texttt{record}(M_{message}(\texttt{lambda}\ (I_{self}\ I_{formal}*)\mathcal{T}[\![E_{body}]\!])))$$

$$\mathcal{T}[\![(\texttt{object-compose}\ E_{obj1}\ E_{obj2})]\!] = (\texttt{override}\mathcal{T}[\![E_{obj2}]\!]\ \mathcal{T}[\![E_{obj1}]\!])$$

$$\mathcal{T}[\![(\texttt{send}\ M\ E_r\ E_1 \ldots\ E_n)]\!] =$$
$$(\texttt{let}\ ((I_{new}\ \mathcal{T}[\![E_r]\!]))$$
$$((\texttt{select} I_r\ M)\ I_r\ \mathcal{T}[\![E_1]\!]\ldots\mathcal{T}[\![E_n]\!]))$$

$$\mathcal{T}[\![N]\!] = (\texttt{letrec}\ ((\texttt{make-integer}(\texttt{lambda}\ (n)(module$$
$$(I_{int} n)$$
$$(+\ldots)\ldots$$
$$(\texttt{make-integer} N))$$