

Bindings

 ⁿ Use let to define (local and global) variables
 ^a add var keyword to allow assignment, otherwise immutable

- n must initialize at declaration
 - let inc := 1;
 let var count := 0;
 - let var count := 0; count := count + inc;

Functions

- $_{\rm n}~$ Use <code>method</code> to define functions
- last expression evaluated is returned
 can overload name for different numbers of arguments

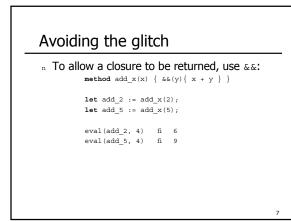
```
let var count := 0;
method foo(a, b, c) {
    count := count + 1;
    let var d := a + b;
    let e := frob(d, c);
    d := d + e;
    d + 5 }
method frob(x, y) { x - frob(y) + 1 }
method frob(x) { - x / 5 }
```

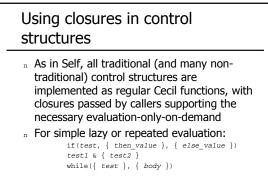
Closures: first-class functions

- n Code in braces is a 0-argument function value let closure := { factorial(10) + 5 };
- n Evaluation of closure delayed until eval is sent: eval(closure) fi 3628805
- n To allow arguments, add & (x,y,z) prefix; invoke passing extra arguments to eval: let closure2 := &(n) { factorial(n) + 5 };
 - eval(closure2, 10) fi 3628805
- " Like ML's fn, Self's blocks
 - $_{\scriptscriptstyle\rm II}$ anonymous, lexically scoped, first-class

5

Glitch: returning closures In current Cecil implementation, by default, closures cannot safely be returned out of their lexically enclosing scope a glitch in the Vortex implementation, not the Cecil language can crash Vortex mysteriously prevents currying, compose, closures in data structures, ...





More examples

- n For iteration with arguments: for(start, stop, &(i) { body }) do(array, &(elem) { body }) do_associations(table, &(key,value) { body })
- n For 3-way branching: compare(i, j, {if_lt}, {if_eq}, {if_gt})

q

11



-- this is a factorial method
method factorial(n) {
 if(n = 0,
 { 1 },
 { n * factorial(n - 1) }) }

-- call factorial here: factorial(7)

Non-local returns

- n Support exiting a method early with a nonlocal return from a nested closure
 n like ^ in Self
 - " like a return statement in C

```
{ ...; ^ result }
{ ...; ^ } -- return void
```

Example method fetch(table, key, if_absent) {

10



n No classes!

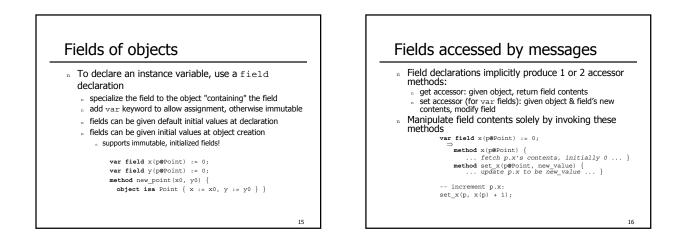
- To make a new "instance" of that ADT, use an object isa ... expression
 - method new_point() {
 object isa Point }
 - n No special constructors!

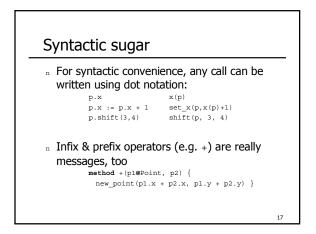
Methods of objects

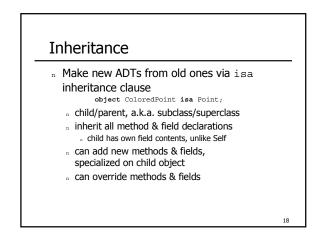
To define a method "in" an object, write the method outside the object but **specialize** the method to the object by adding @obj after the first argument (which acts like the receiver argument)

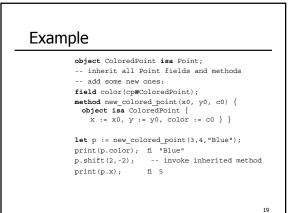
```
method area(p@Point) {
    p.x * p.y }
method shift(p@Point, dx, dy) {
    p.x := p.x + dx;
    p.y := p.y + dy; }
```

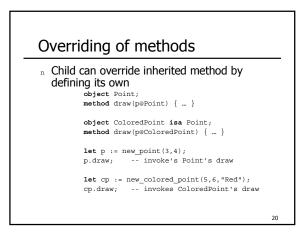
14

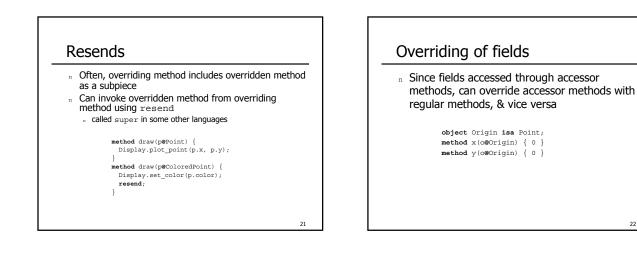












Accessing fields

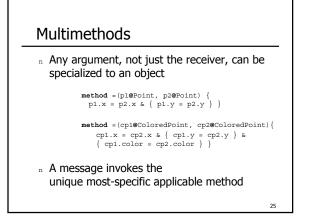
- Because fields accessed through messages, like methods, clients can't tell how message implemented
 - n can differ in different child objects
 - can change through program evolution & maintenance

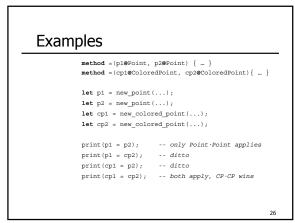
let p := ...; -- Point or Origin object
print(p.x); -- how is x implemented?

23

Overloaded methods and dynamic dispatching

- ⁿ Can overload methods two ways:
 - a same name but different numbers of arguments
 a same name & number of arguments, but different specializer objects
- Specializer-based overloading resolved by using run-time class of receiver argument (a.k.a. dynamic dispatching, message sending)
 - a unlike static overloading, which uses only the static type known at the call site

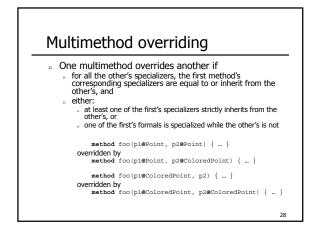




Method lookup rules

- Find all methods with the right name and number of arguments that apply
 - A method applies if the actual run-time objects are equal to or inherit from all the method's specializers, where present
- Report "message not understood" if no applicable methods
 Pick the applicable method whose specializers are uniformly most specific
 - A specializer is more specific than another if it inherits from the other
 - A method overrides another if all of its specializers are at least as specific as the other's
 - Report "message ambiguous" if no single best method

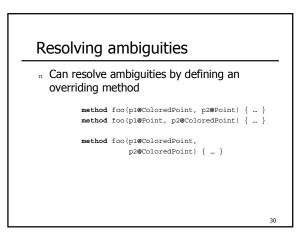
27

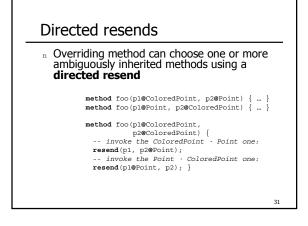


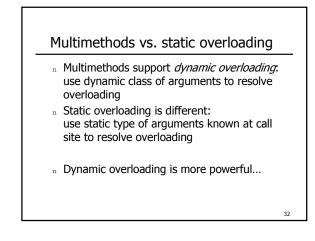
Ambiguous methods

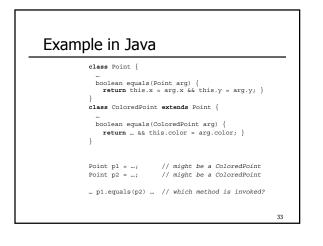
 Two methods may be mutually ambiguous: neither overrides the other

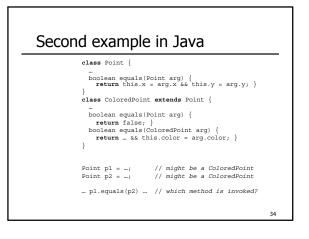
```
method foo(p1@Point, p2) { ... }
ambiguous with
  method foo(p1, p2@Point) { ... }
  method foo(p1@ColoredPoint, p2@Point) { ... }
ambiguous with
  method foo(p1@Point, p2@ColoredPoint) { ... }
```

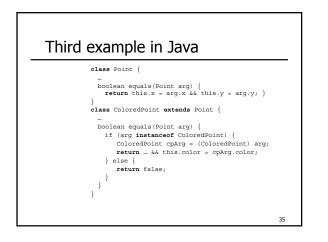


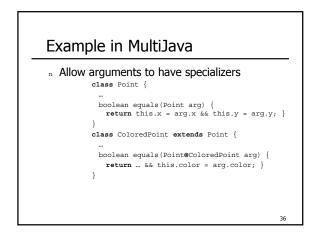












Some uses for multimethods

- ⁿ Multimethods useful for binary operations
- ¹ 2+ arguments drawn from some abstract domain with several possible implementations
- ⁿ Examples:
 - ⁿ equality over comparable types
 - $_{\rm n}\,$ <, >, etc. comparisons over ordered types
 - n arithmetic over numbers
 - $_{\rm n}\,$ union, intersection, etc. over set representations

Some more uses

 ${\scriptstyle \rm n}$ Multimethods useful for cooperative operations even over different types

n Examples:

- $_{\rm n}~{\rm display}$ for various kinds of shapes on various kinds of output devices
 - " standard default implementation for each kind of shape
- overridden with specialized implementations for certain devices
 handleEvent for various kinds of services for various kinds of events
- operations taking flag constant objects, with different algorithms for different flags

Advantages of multimethods

- ⁿ Unify & generalize:
 - top-level procedures (no specialized arguments)
 regular singly-dispatched methods (specialize first argument)
 - a overloaded methods (resolve overloading dynamically, not statically)
- ⁿ Naturally allow existing objects/classes to be extended with new behavior
- n Avoid tedium & non-extensibility of instanceof/cast

Challenges of multimethods

- Dbjects don't contain their methods, so...
 What's the programming model?
 What's the encapsulation model?
- How to typecheck definitions and calls of multimethods?
- ⁿ How to implement efficiently?

Multiple inheritance

ⁿ Can inherit from several parent objects:

object Shape; object Rectangle isa Shape; object Rhombus isa Shape; object Square isa Rectangle, Rhombus;

object Stream; object InputStream isa Stream; object OutputStream isa Stream; object IOStream isa InputStream, OutputStream;

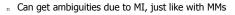
- ⁿ MI can be natural in application domain
- MI can be useful for better factoring & reuse of code
 But MI introduces semantic complications....

41

37

39

Ambiguities



object Rectangle isa Shape; method area(r@Rectangle) { ... } object Rhombus isa Shape; method area(r@Rhombus) { ... } object Square isa Rectangle, Rhombus;

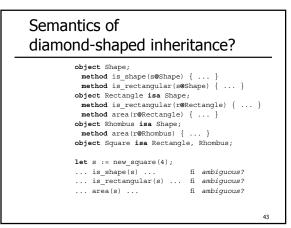
let s := new_square(4);
... area(s) ... fi ambiguous!

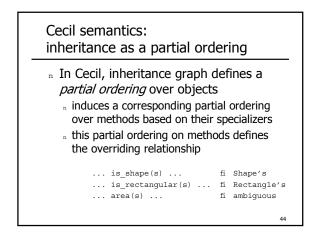
 Can resolve ambiguities by adding overriding method, just as with MMs

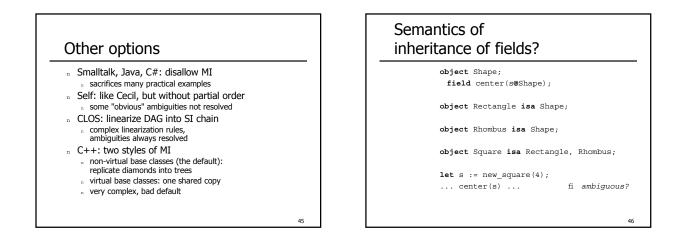
method area(s@Square) { resend(s@Rectangle) }

42

38

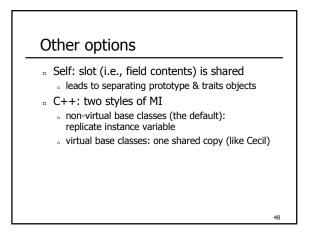






Cecil semantics: fields are shared

- In Cecil, fields are present once, independently of along how many paths they are inherited
 - field accessor methods are treated just like regular methods
 - $\ensuremath{\,^{\ensuremath{\scriptscriptstyle n}}}$ field contents are stored once per inheriting object
 - ... center(s) ... fi s's contents of Shape's center field



Mixins

- n
- MI enables new programming idioms, including *mixins*: highly factored abstract objects Typically, organize attributes along independent axes n
- several possible implementations (mixins) for each axis each concrete subclass picks one mixin for each axis
- each concrete subclass picks one mixin for each axis
 Example axes for shapes in a user interface:

 colored or not, bordered or not, titled or not, mouse-click handler,...
 Different mixin axes have common parent (e.g. Shape), leading to diamond-shaped inheritance

object CheckBox isa Square, BorderedShape, ClickableShape, ...;

Java's approach

- ⁿ Java supports two flavors of classes: regular classes and interfaces
- ⁿ Interfaces include no implementation, just "abstract methods"
 - n no instance variables
 - n no method bodies
- ⁿ Allow multiple inheritance only of interfaces n a class can inherit from at most one regular class
 - n an interface can inherit only from interfaces

50

Analysis of Java's approach

- ⁿ Benefits:
 - n no method bodies in interfaces ⇒
 no ambiguities between implementations
 - $_{n}$ no instance variables in interfaces \Rightarrow
 - no ambiguities in instance variable offset calculations
 - ⁿ still support some multiple inheritance idioms n primarily for static type checking, not code reuse
- n Costs:
- n no mixin-style programming
- additional language complexity and library size

51