

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;

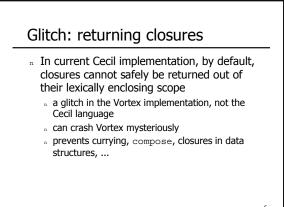
Functions

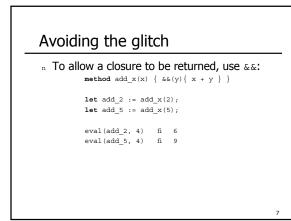
- $_{\rm n}~$ Use <code>method</code> to define functions
- last expression evaluated is returned
 can overload name for different numbers of arguments

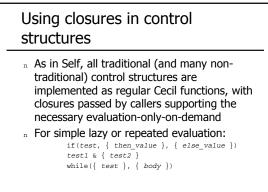
```
let var count := 0;
method foo(a, b, c) {
    count := count + 1;
    let var d := a + b;
    let e := frob(d, c);
    d := d + e;
    d + 5 }
method frob(x, y) { x - frob(y) + 1 }
method frob(x) { - x / 5 }
```

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
- ⁿ Like ML's fn, Self's blocks
 - n anonymous, lexically scoped, first-class







More examples

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

q

11



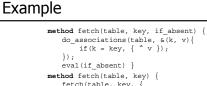
-- this is a factorial method method factorial(n) { if(n = 0,{ 1 }, $\{ n * factorial(n - 1) \} \}$

-- 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
 - ⁿ like a return statement in C

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



fetch(table, key, {
 error("key " ||
 print_string(key) ||
 " not found") }) }

fetch(zips, "Seattle", { 98195 })

2

12



n No classes!

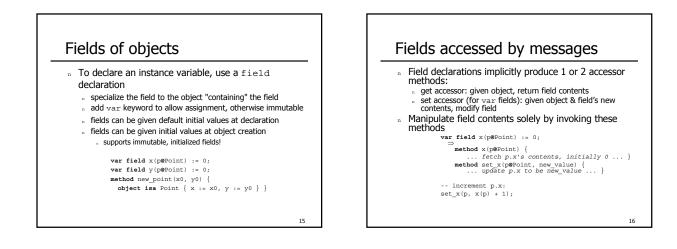
- To make a new "instance" of that ADT, use an object isa ... expression
 - method new_point() {
 object isa Point }
 - n No special constructors!

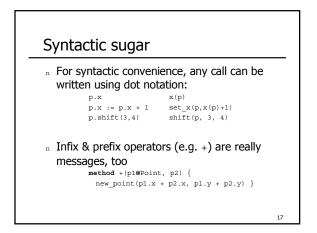
Methods of objects

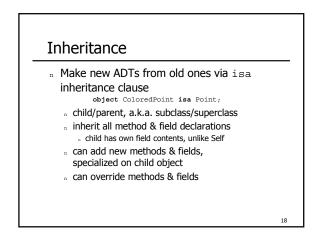
To define a method "in" an object, write the method outside the object but **specialize** the method to the object by adding @obj after the first argument (which acts like the receiver argument)

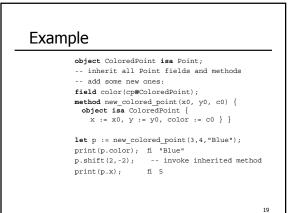
```
method area(p@Point) {
    p.x * p.y }
method shift(p@Point, dx, dy) {
    p.x := p.x + dx;
    p.y := p.y + dy; }
```

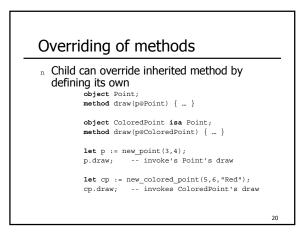
14

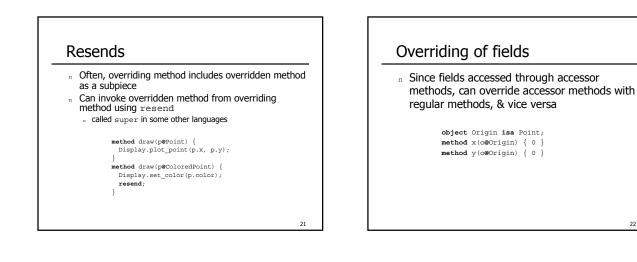












Accessing fields

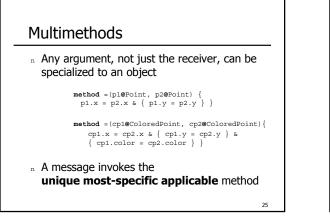
- Because fields accessed through messages, like methods, clients can't tell how message implemented
 - n can differ in different child objects
 - can change through program evolution & maintenance

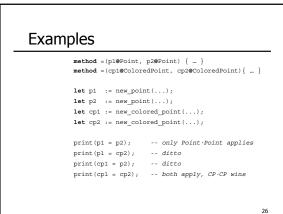
let p := ...; -- Point or Origin object
print(p.x); -- how is x implemented?

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





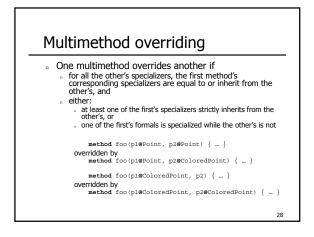
Method lookup rules

- ⁿ 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

- $\ensuremath{\,^{\rm n}}$ 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

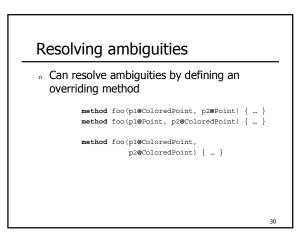
27

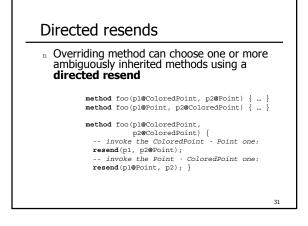


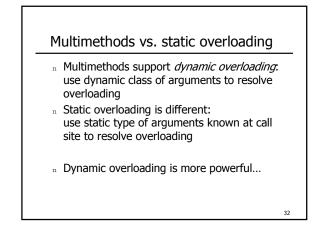
Ambiguous methods

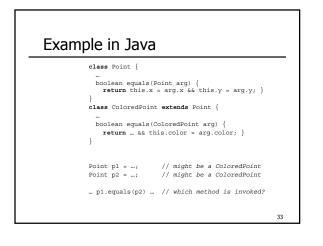
n Two methods may be mutually ambiguous: neither overrides the other

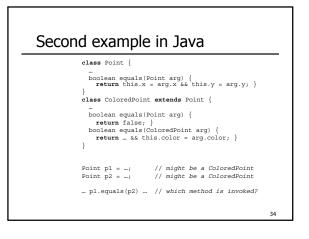
```
method foo(p1@Point, p2) { ... }
ambiguous with
  method foo(p1, p2@Point) { ... }
  method foo(p1@ColoredPoint, p2@Point) { ... }
ambiguous with
  method foo(p1@Point, p2@ColoredPoint) { ... }
```

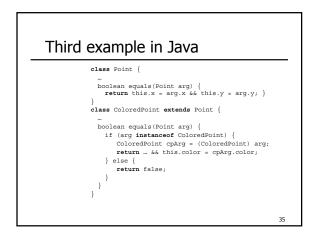


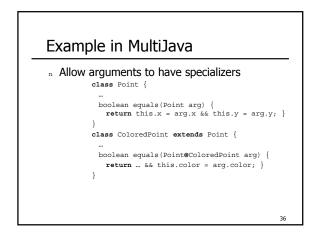












Some uses for multimethods

- 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?
- n 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;

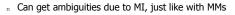
- ⁿ MI can be natural in application domain
- MI can be useful for better factoring & reuse of code
 But MI introduces semantic complications....

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;

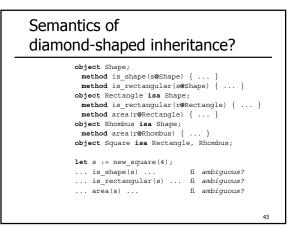
let s := new_square(4);
... area(s) ... fi ambiguous!

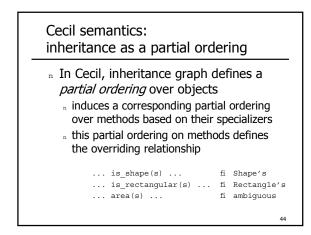
 Can resolve ambiguities by adding overriding method, just as with MMs

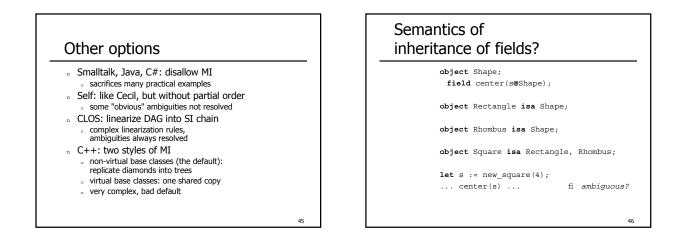
method area(s@Square) { resend(s@Rectangle) }

42

38

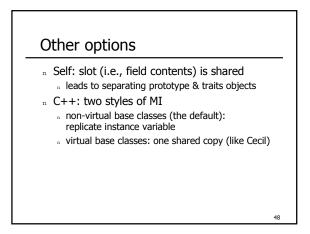






Cecil semantics: fields are shared

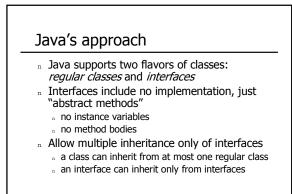
- In Cecil, fields are present once, independently of along how many paths they are inherited
 - field accessor methods are treated just like regular methods
 - $\ensuremath{\,^{\ensuremath{\scriptscriptstyle n}}}$ field contents are stored once per inheriting object
 - ... center(s) ... fi s's contents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



Mixins

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



50

52

54

Analysis of Java's approach

- n Benefits:
 - $_{n}$ no method bodies in interfaces \Rightarrow 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

Typechecking OO Languages

- ⁿ In OO language, want static checking to ensure the absence of: n message-not-understood errors ⁿ message-ambiguous errors
- ⁿ Want to allow subclasses to be used in place of superclasses n as long as this doesn't create errors

General strategy

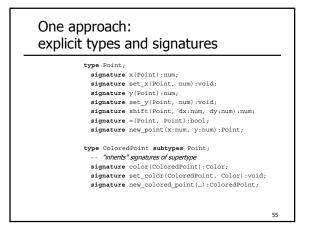
- ⁿ Declare (or infer) **types** and their **subtyping** relationships
- ⁿ Declare (or infer) types of variables
- Check that assignments/initializations/returns only store subtypes of variable's type
- ⁿ Declare **signatures** of operations Check that messages with particular actual
 - argument types find at least one matching signature
 - Check that methods & fields completely and unambiguously implement covering signatures

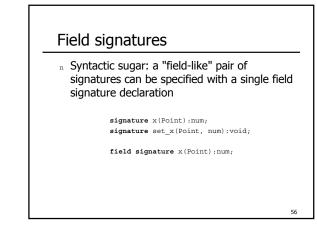
53

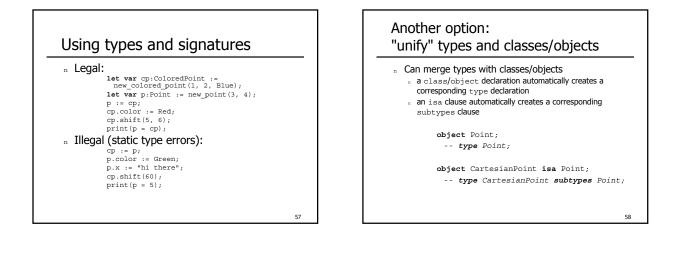
51

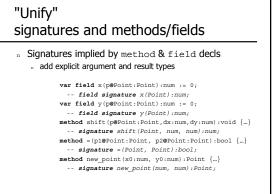
Points of variation

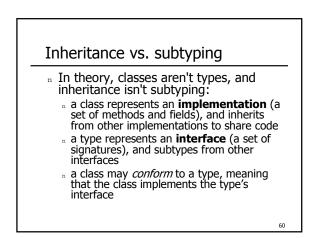
- ⁿ What's a type?
- ⁿ What's a subtype?
- ⁿ What's a signature?





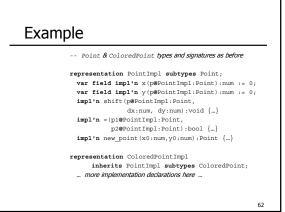






Cecil's approach

- ⁿ In Cecil, can program these separately:
 - n type, subtypes, signature declarations for interfaces
 - n representation, inherits, implementation declarations for implementation
 - subtypes declarations to conformance of implementations to interfaces

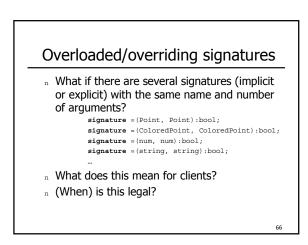


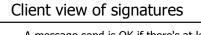
Benefits of separating Syntactic sugar inheritance and subtyping ⁿ Clarity of thinking ⁿ Common case: ⁿ Sensible to implement interface w/o inheriting inheritance and subtyping are parallel code n object defines representation & type ⁿ Akin to Java's interfaces " the representation subtypes the type ⁿ Sometimes sensible to reuse code w/o being n isa defines parallel inherits & subtypes a subtype method defines implementation & signature E.g. if ColoredPoint wants to inherit Point's code, but not allow ColoredPoints to mix with uncolored . @: does parallel @ and : Points method = (pl@Point:Point, p2@Point:Point) :... ⁿ Sometimes the two relations are opposite object deque subtypes stack; object stack inherits deque; method = (p1@:Point, p2@:Point):bool {...} 63 64

61

Costs of separating inheritance and subtyping

- $_{\rm n}~$ Verbosity in common case
 - $n \Rightarrow$ need syntactic sugar
- n Complexity
- Subtyping w/o inheritance cannot provide default implementations
 - n A weakness of Java's interfaces
- $\ensuremath{\,^{\ensuremath{n}}}$ Difficult to typecheck safety of inheriting w/o subtyping





- $\ensuremath{\,^{\ensuremath{\scriptscriptstyle n}}}$ A message send is OK if there's at least one signature that says so
 - n **E.g.** cpl = cp2

is legal if there's some signature whose argument types are (supertypes of) ColoredPoint

ⁿ The client doesn't have to "choose" the right one, or do dispatching

Legality

- ⁿ To make signatures legal, whatever promises they make to clients have to be guaranteed by method implementations
- If a client could pass certain types of arguments in a message, then
 - exactly one method has to be able to handle those arguments
 - $\ _{\rm n}$ the result type of the method has to be something that the client will expect
- Related to when one method can legally override another

68

70

Legality of method overriding

- Sufficient condition for safety: overriding method has same argument and result types as overridden method
 - ensures that using signature from originating method in checking calls won't be broken if overriding method selected at run-time
- n Are relaxed conditions also safe?
 - n can the result type be more precise (or more general) in overriding method?
 - n can an argument type be more precise (or more general) in overriding method?

69

67

An example

n Which (if any) of the overrides are legal? method copy(p@:Point):Point method copy(p@:ColoredPoint):ColoredPoint method copy(p@:Point3D):Object

let p:Point := ...; -- a Point, ColoredPt, or Point3D
let q:Point := p.copy;
... q.x ...

let cp:ColoredPoint := ...; -- a ColoredPoint
let cq:ColoredPoint := cp.copy;
... cq.color ...

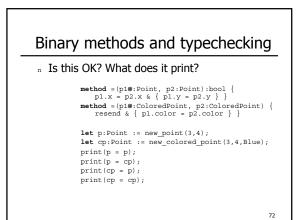
Another example

n Which (if any) of the overrides are legal? method slide(p@:Point, dx:num):void method slide(p@:ColoredPoint, dx:int):void method slide(p@:Point3D, dx:Object):void

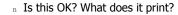
let p:Point := ...; -- a Point, ColoredP't, or Point3D
slide(p, 3.4);

let cp:ColoredPoint := ...; -- a ColoredPoint
slide(cp, 5);

let p3d:Point3D := ...; -- a Point3D
slide(p3d, "hi");

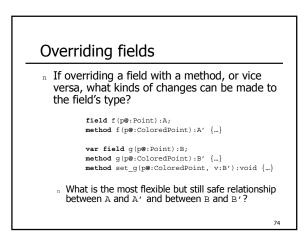


Binary methods with multimethods



```
method =(p1@:Point, p2@:Point):bool {
    p1.x = p2.x & { p1.y = p2.y } }
method =(p1@:ColoredPoint, p2@:ColoredPoint){
    resend & { p1.color = p2.color } }
```

```
let p:Point := new_point(3,4);
let p:Point := new_colored_point(3,4,Blue);
print(p = p);
print(p = cp);
print(cp = p);
print(cp = cp);
```



Summary of overriding

- ⁿ Legal to override method in subtype if:
 - ⁿ result type same or a subtype
 - (covariant)
 - argument types same or supertypes
 - (contravariant)
 - n for undispatched argumentsn dispatched arguments are replaced with subtypes
- Contravariance is a pain in practice, but
 "It's the Law" (for type safety, at least)

75

73

Checking signatures

- In Cecil, allow arbitrary signatures and implementations
- Need to ensure that each signature is completely and unambiguously implemented by one or more methods
- n Naïve algorithm:
 - foreach combination of classes of arguments which is typecorrect according to the signature
 - . do method lookup
 - » verify unique most-specific applicable method found
- n Efficiency? Modularity?

Abstract classes and methods

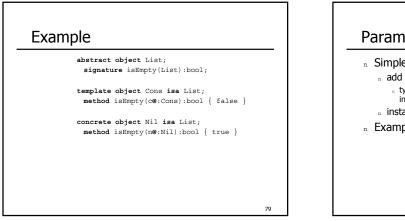
- Most OO languages allow abstract classes, which can have abstract (unimplemented) methods
 - Abstract methods OK as long as no instances of the abstract class can be created
- n Cecil supports this idea through object role annotations
 - . Used only during typechecking

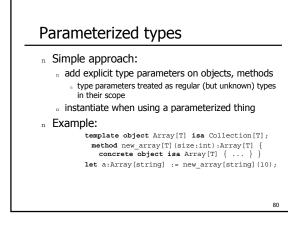
77

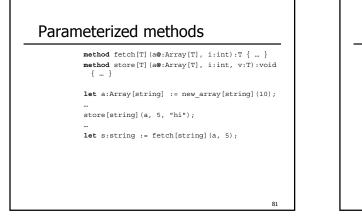
Object roles abstract object: like an abstract class cannot be manipulated directly doesn't have to have its signatures implemented template object: like a concrete class cannot be manipulated directly has to have its signatures implemented concrete class

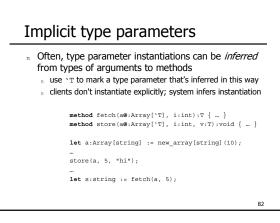
- n concrete object: like an instance
 - n can be manipulated directly has to have its signatures implied
 - $_{\rm n}\,$ has to have its signatures implemented

78



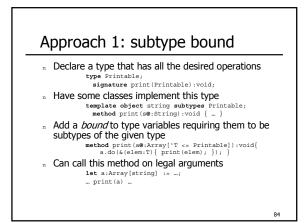


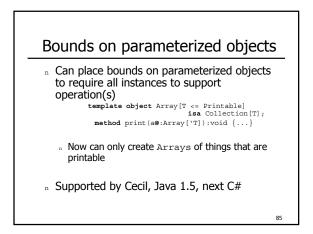


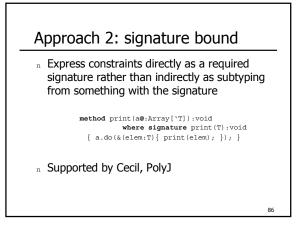


Universal vs. bounded parametric polymorphism

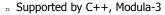
- Mant to place constraints on legal instantiations of type variables, so that we can do interesting things with values of that type
 - . ML has equality types
 - Wish ML had more flexible kinds of type for things that support print, <, etc.</p>
- n Example:
 - a print method on Array[T],
 - given that elements can be printed
 - " how to express the constraint on ${\mathbb T}$ such that values of type ${\mathbb T}$ are known to be printable?





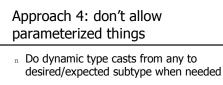








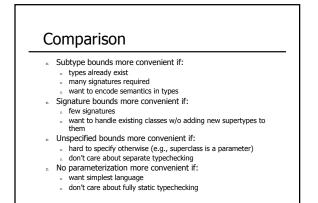
89



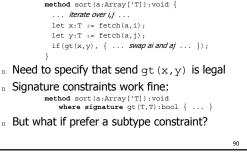
method print(a@:Array):void {
 a.do(&(elem:any) {
 let e:Printable := cast[Printable](elem);
 print(e);
 });
}

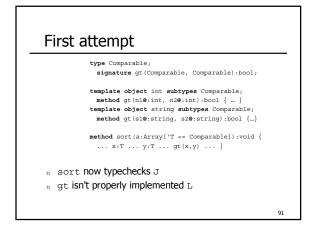
88

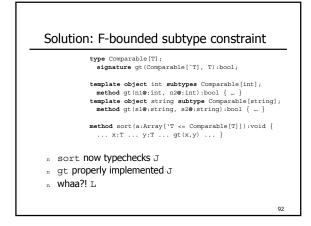
ⁿ Java 1.4 and earlier, current C#



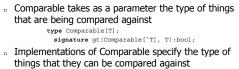
Polymorphism over binary methods







In English...



- object int subtypes Comparable[int]; object string subtypes Comparable[string];
- Sort takes an array of things that can be compared against themselves

method sort(a:Array['T <= Comparable[T]]):void {...}</pre>

93

Another example

method max(x:`T, y:`T):T
 where T <= Comparable[T]
 { if(gt(x,y), { x }, { y }) }</pre>

- n max on strings returns a string
- $n \max$ on ints returns an int
- $\tt n$ a static type error to try to do $\tt max$ on a string and a number

Summary

- F-bounded polymorphism is required for many practical examples of OO polymorphism
 Supported in Cecil, Java 1.5, new C#
- ⁿ Pretty tricky to learn how to define your own F-bounded classes and methods
- $\ensuremath{\,^{\ensuremath{\scriptstyle n}}}$ Signature-bounded polymorphism remains simple

95