To inherit or not to inherit

Inheritance works well when

- subclass supports a superset of operations of superclass
  ⇒ can use subclass where superclass expected
- subclass reuses much of implementation of superclass
- subclass's representation extends rep of superclass
- subclass is a special kind of superclass, conceptually
  - subclass "is a" superclass

Inheritance is not appropriate when:

- class has another class as a component
  - class "has a" class
  - interfaces are not related
  - use a slot instead
- only part of other class's implementation is reused
- representation of other class needs to be altered

If in doubt, don't inherit,
or reorganize class hierarchy to make inheritance work better

Inheritance vs. forwarding

Forwarding:

- when invoke operation on object *A*, forward the operation to component object *B*

This is not inheritance:

- true inheritance remembers the original object, so that "sends to self" work (go back to *A*, not *B*)

Examples of multi-methods

Multi-methods useful for binary operations

- 2+ arguments drawn from some abstract domain with several possible implementations

Examples:

- equality over comparable types
- < etc. comparisons over ordered types
- arithmetic over numbers
- `union`, `intersection`, etc. over set representations

Multi-methods useful for cooperative operations even over different types

Examples:

- `display` for different kinds of shapes on different kinds of output devices
  - standard default implementation for each kind of shape
  - overridden with specialized implementations for certain devices
- operations taking flag constant objects, with different algorithms for different flags

Typechecking OO Languages

In OO language, want static checking to ensure the absence of:

- message-not-understood errors
- message-ambiguous errors

[In a purely OO language, that's all that typechecking does]

Want to allow subclasses to be used in place of superclasses, as long as doesn't create errors like the above

Idea:

- declare (or infer) "types" of variables, "signatures" of operations
- check that assignments/initializations/returns only store "subtypes" of variable's type
- check that messages with particular actual argument types find at least one matching signature, guaranteeing the msg is understood unambiguously and specifying msg result type
- check that methods & fields completely and unambiguously implement covering signatures

Key questions: what's a type? subtype? signature?

Simple idea: type = class

In many OO languages, types and classes/objects are the same

- subtyping = inheritance
- signature = derived directly from `method` & `field` decls, and from explicit `signature` decls
  - `signature` decls replace abstract method decls
- overriding methods keep same argument & result types, for unspecialized formals

```
abstract object Point; -- Point is a type too
  signature x(p:Point):num;
  signature y(p:Point):num;
  method area(p@:Point):num { p.x * p.y }
  method rotate(p@:Point, amount:num):Point{.}
  method =(p1@:Point, p2@:Point):bool {
    p1.x = p2.x & { p1.y = p2.y } }

template object CartesianPoint isa Point;
  var field x(p@:CartesianPoint):num := 0;
  var field y(p@:CartesianPoint):num := 0;
  method new_cartesian_point(x0:num, y0:num
                            ):CartesianPoint {
    concrete object isa CartesianPoint ... }
```


Interfaces

Previous declarations imply the following interfaces

- can also be declared explicitly if desired

```
type Point;
  signature x(Point):num;
  signature y(Point):num;
  signature area(Point):num;
  signature rotate(Point, amount:num):Point;
  signature =(Point, Point):bool;

type CartesianPoint subtypes Point;
  -- no need to add x & y signatures for CartesianPoint
  signature set_x(CartesianPoint, num):void;
  signature set_y(CartesianPoint, num):void;
  signature new_cartesian_point(num, num
                               ):CartesianPoint;
```

Use these declarations for typechecking
initializations, assignments, and message sends

Examples

Legal:

```
let var p1:CartesianPoint :=
  new_cartesian_point(1,2);
let var p2:Point := new_cartesian_point(3,4);
p2 := p1;
p1.x := 20;
print(p1 = p2);
p2 := p2.rotate(60);
```

Illegal (static type error):

```
p2.x := 20;
p1 := p2;
p1.x := "hi there";
p1 := p2.rotate(60);
```


Method compatibility

Sufficient condition for safety:
an overriding method must have same argument and result types

- ensures that using signature from originating method in checking calls won't be broken if overriding method selected at run-time

Relaxed conditions (**contravariance**):

- result type can be more precise in overriding method
- argument type can be more general in overriding method

Example: is this OK? under what rules?

```
signature x(Point):num;
signature set_x(Point, num):void;
...
var field x(p@:CartesianPoint):int := 0;
  →
    signature x(CartesianPoint):int;
    signature set_x(CartesianPoint, int):void;
```

Binary methods and typechecking

Another example (note single dispatching, as in most OOLs); is this OK?

```
abstract object Point;
method =(p1@:Point, p2:Point):bool {
  p1.x = p2.x & { p1.y = p2.y } }

abstract object ColoredPoint isa Point;
method =(p1@:ColoredPoint, p2:ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Example of use:

```
let p1:Point := new_cartesian_point(3,4);
let p2:Point := new_colored_point(3,4,"Blue");
print(p1 = p1);
print(p1 = p2);
print(p2 = p1);
print(p2 = p2);
```


Binary methods with multi-methods

Another example (note multiple dispatching); is this OK?

```
abstract object Point;
method =(p1@:Point, p2@:Point):bool {
  p1.x = p2.x & { p1.y = p2.y } }

abstract object ColoredPoint isa Point;
method =(p1@:ColoredPoint, p2@:ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Example of use:

```
let p1:Point := new_cartesian_point(3,4);
let p2:Point := new_colored_point(3,4,"Blue");
print(p1 = p1);
print(p1 = p2);
print(p2 = p1);
print(p2 = p2);
```

What semantics for mixed colored & plain points is desired?