

Types and Classes

I

- From predefined (simple) and user-defined (composite) types
 - via
- Abstract data types
 - to
- Classes
 - Type compatibility
 - Subtyping <> subclassing
 - Class compatibility
 - Covariance/contravariance
 - Types of parameters of redefined methods

II

- Advanced oo concepts
 - Specialization of behaviour?
 - Multiple inheritance - alternatives
 - Inner classes
- Modularity
 - Packages
 - Interface-implementation
- Generics
- Scandinavian approach to object oriented programming

Classification of types

- Predefined, simple types (not built from other types)
 - boolean, integer, real, ...
 - pointers, pointers to procedures
 - string
 - User-defined types
 - enumerations
 - Predefined composite types
 - arrays
 - User-defined, composite types
 - Records/structs, unions, abstract data types, classes
-
- Evolution from simple types, via predefined composite types to user-defined types that reflect parts of the application domain.

Properties of primitive types

- Classifying data of the program
- Well-defined operations on values
- Protecting data from un-intended operations
- Hiding underlying representation

Properties of composite types

- Records, structs (Cartesian products)
 - (m_1, m_2, \dots, m_n) in $M_1 \times M_2 \times \dots \times M_n$
 - Assignment, comparison
 - Composite values $\{3, 3.4\}$
 - Hiding underlying representation?

```
typedef struct {  
    int nEdges;  
    float edgeSize;  
} RegularPolygon;  
RegularPolygon rp = {3, 3.4}  
rp.nEdges = 4;
```

- Arrays (mappings)
 - domain \rightarrow range
 - Possible domains, index bound checking, bound part of type definition, static/dynamic?

```
char digits[10]  
  
array [5..95] of integer  
  
array[WeekDay] of T,  
where  
type WeekDays =  
    enum{Monday, Tuesday, ...}
```

Composite types

- Union
 - Run-time type check

```
union address {  
    short int offset;  
    long int absolute; }  
}
```

```
union bad_idea {  
    int int_v;  
    int* int_ref; }  
}
```

- Discriminated union
 - Run-time type check

```
address_type = (absolute, offset);  
  
safe_address =  
record  
    case kind:address_type of  
        absolute: (abs_addr: Integer);  
        offset: (off:addr: short integer)  
end;
```

```
typedef struct {  
    address location;  
    descriptor kind;  
} safe_address;  
  
enum descriptor {abs, rel}
```

Type compatibility (equivalence)

- Name compatible
 - Values of types with the same name are compatible
- Structural compatible
 - Types T1 and T2 are structural compatible
 - If T1 is name compatible with T2, or
 - T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components

```
struct Position { int x,y,z; };  
struct Position pos;  
struct Date { int m,d,y; };  
struct Date today;  
  
void show(struct Date d);  
  
...; show(today); ...  
  
...; show(pos); ...
```

```
struct Complex { real x,y; };  
  
struct Point {real x,y; };
```

Abstract datatypes

```
abstype Complex = C of real * real
  with
    fun complex(x,y: real) = C(x,y)
    fun add(C(x1,y1),C(x2,y2)) = C(x1+x2,y1+y2)
  end

...; add(c1,c2); ...
```

Signature

- Constructor
- Operations

Abstract datatypes versus classes

```
abstype Complex = C of real * real
  with
    fun complex(x,y:real) = C(x,y)
    fun add(C(x1,y1,C(x2,y2)) = C(x1+x2,y1+y2)
  end
```

```
...; add(c1,c2); ...
```

```
class Complex {
  real x,y;
  Complex(real v1,v2) {x=v1; y=v2}
  add(Complex c) {x=x+c.x; y=y+c.y}
}
```

```
...; c1.add(c2); ...
```

With abstract data types:
operation (operands)

⇒ meaning of operation is always
the same

With classes
object.operation (arguments)

⇒ code depends on
object and operation
(dynamic lookup, method dispatch)

Possible to do 'add(c1,c2)' with classes?

From abstract data types to classes

- Encapsulation through abstract data types
 - Advantage
 - Separate interface from implementation
 - Guarantee invariants of data structure
 - only functions of the data type have access to the internal representation of data
 - Disadvantage
 - Not extensible in the way that OO is

Subtyping (in general)

- Subtype as a subset of the set of values
 - e.g. subrange
- Compatibility rules between subtype and supertype
 - Substitutability principle: a value of a subtype can appear wherever a value of the supertype is expected

Abstract data types argument of Mitchell

```
abstype queue
with
    mk_Queue: unit -> queue
    is_empty: queue -> bool
    insert: queue * elem -> queue
    remove: queue -> elem
is ...
in
    program
end
```

```
abstype pqueue
with
    mk_Queue: unit -> pqueue
    is_empty: pqueue -> bool
    insert: pqueue * elem -> pqueue
    remove: pqueue -> elem
is ...
in
    program
end
```

Cannot apply queue code to pqueue, even though signatures are identical

Point and ColorPoint

```
class Point {  
  int x,y;  
  move(int dx,dy) {  
    x=x+dx; y=y+dy}  
}
```

```
class ColorPoint extends Point {  
  Color c;  
  changeColor(Color nc) {c= nc}  
}
```

Point

x
y
move

ColorPoint

x
y
c
move
changeColor

◆ ColorPoint interface contains Point

- ColorPoint is a *subtype* of Point

Could not form list of points and colored points if done by abstract data types

Object Interfaces - Subtyping

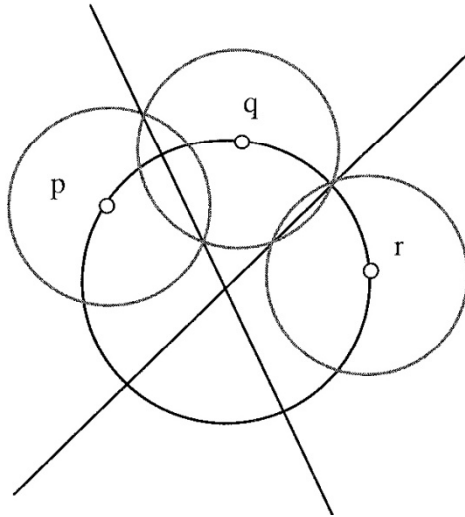
- Interface
 - The operations provided by objects of a certain class
- Example: Point
 - `x` : returns x-coordinate of a point
 - `y` : returns y-coordinate of a point
 - `move` : method for changing location
- The interface of an object is its *type*.
- If interface `B` contains all of interface `A`, then `B` objects can also be used as `A` objects (substitutability)

- Subclassing <> subtyping

Subclassing

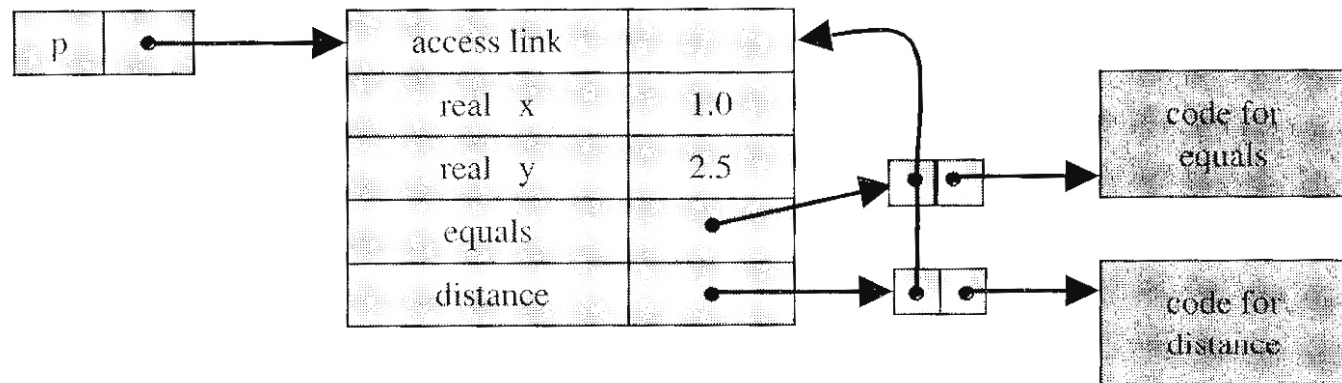
- Two approaches
 - So-called Scandinavian/Modeling Approach
 - Classes represent concepts from the domain
 - Subclasses represent specialized concepts
 - Overriding is specialization/extension
 - Subclass is subtype
 - Reluctant to multiple inheritance (unless it can be understood as multiple specialization)
 - So-called American/Programming Approach
 - Classes represent implementations of types
 - Subclasses inherit code
 - Overriding is overriding
 - Subclassing not necessarily the same as subtyping
 - Multiple inheritance as long as it works

Simula I




```

class Point(x,y); real x,y;
begin
  boolean procedure equals(p); ref(Point) p;
    if p =/= none then
      equals := abs(x - p.x) + abs(y - p.y) < 0.00001;
  real procedure distance(p); ref(Point) p;
    if p == none then error else
      distance := sqrt(( x - p.x )**2 + (y - p.y) ** 2);
end ***Point***
p :- new Point(1.0, 2.5);
  
```



Simula II

```
class Line(a,b,c); real a,b,c;
begin
  boolean procedure parallelto(l); ref(Line) l;
    if l /= none then
      parallelto := abs(a*l.b - b* l.a) < 0.00001;
  ref(Point) procedure meets(l); ref(Line) l;
  begin real t;
    if l /= none and ~parallelto(l) then
      begin
        t := 1/(l.a * b - l.b * a);
        meets :- new Point(..., ...);
      end;
  end; ***meets***
  real d;
  d := sqrt(a**2 + b**2);
  if d = 0.0 then error else
    begin
      d := 1/d;
      a := a * d; b := b * d; c := c * d;
    end;
  end *** Line***
```



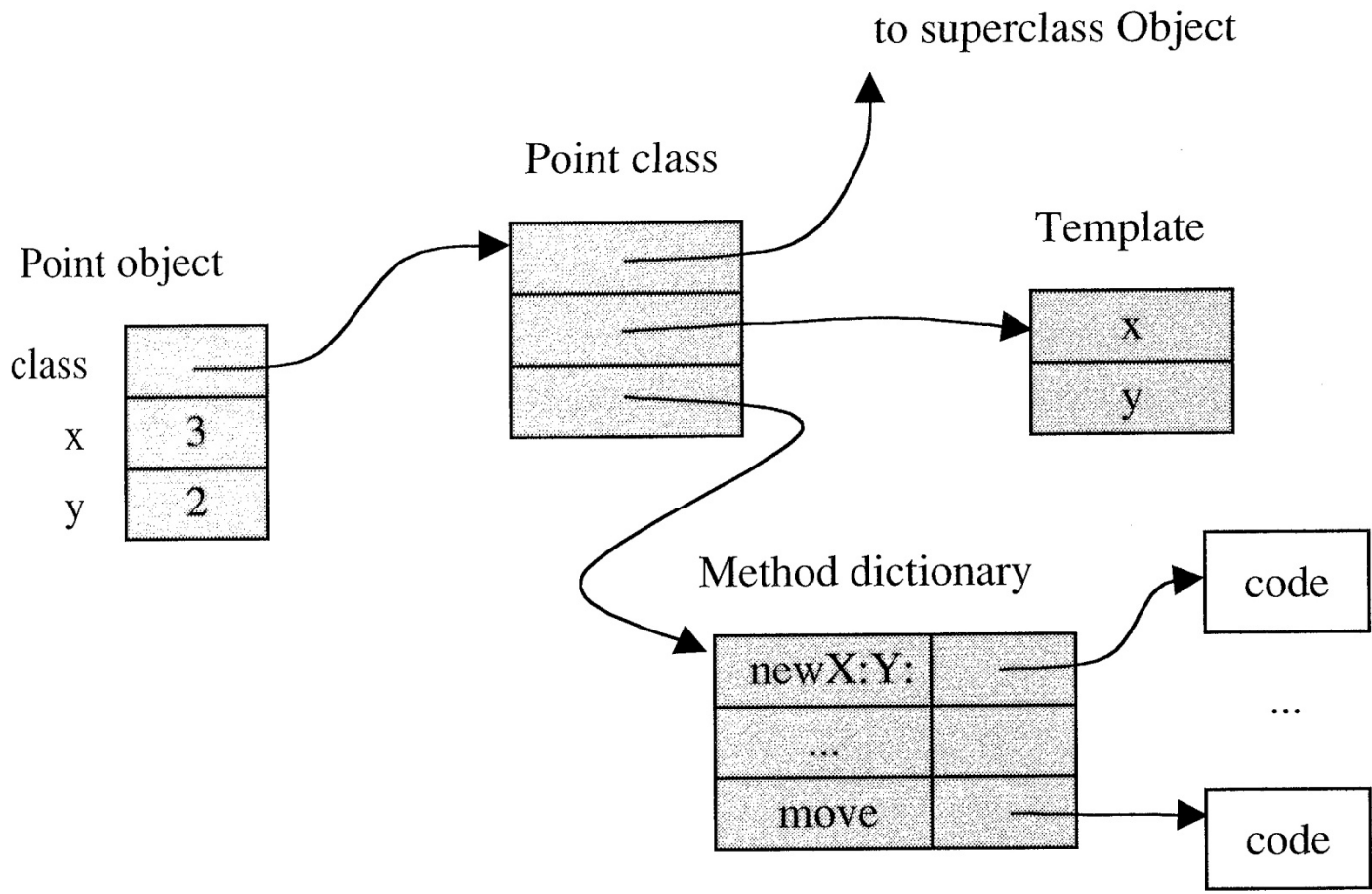
Subclassing in Simula

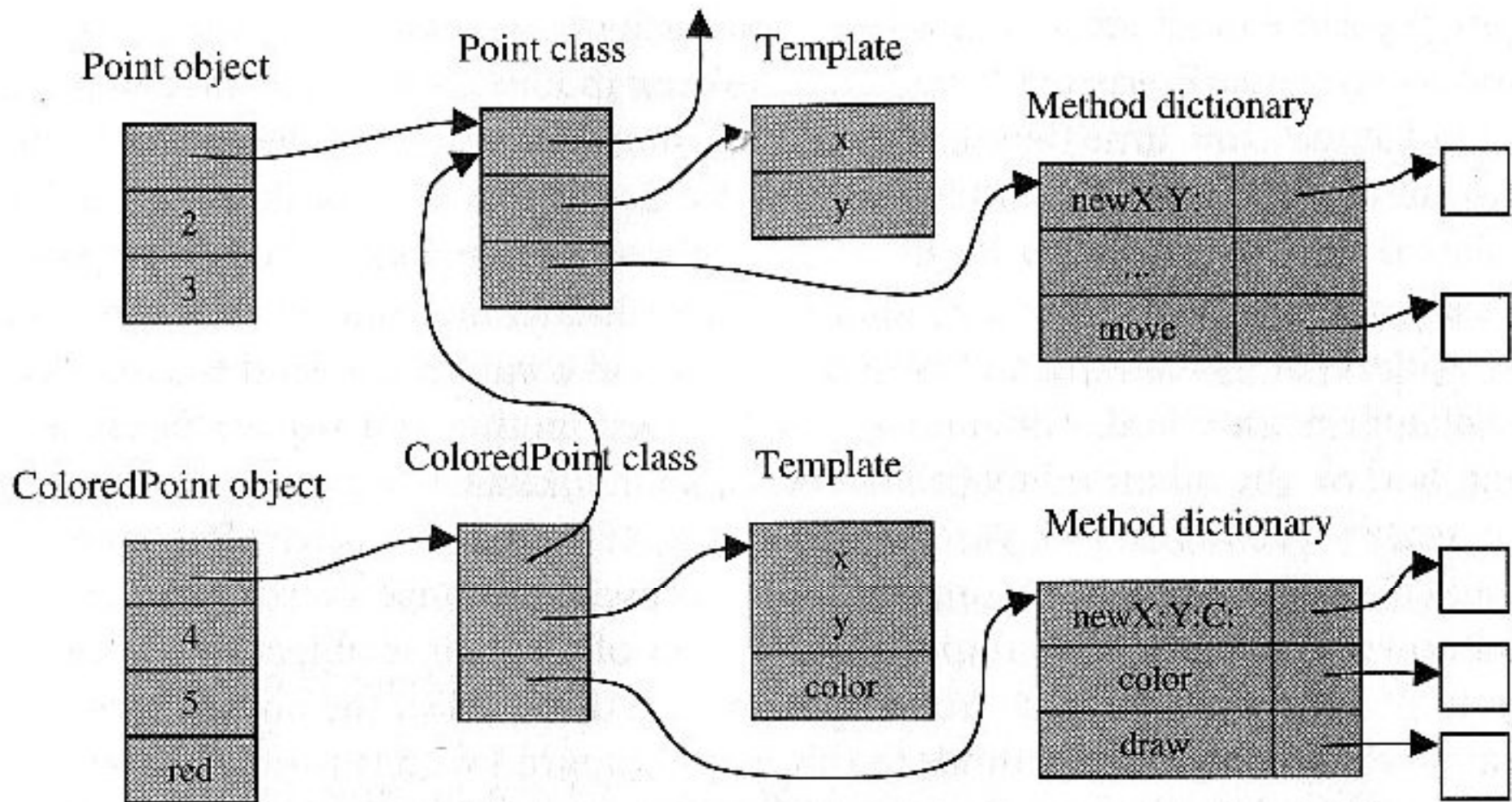
```
Point class ColorPt(c); color c;      ! List new parameter only
begin
  boolean procedure equals(q); ref(ColorPt) q;
  ...;
end ***ColorPt***
ref(Point) p;                          ! Class reference variables
ref(ColorPt) cp;
p := new Point(2.7, 4.2);
cp := new ColorPt(3.6, 4.9, red);      ! Include parent class parameters
```

```
class A;
A class B; /* B is a subclass of A */
ref (A) a;
ref (B) b;
a :- b;    /* legal since B is a subclass of A */
...
b :- a;    /* also legal but checked at run-time to make sure a points to a B
            object*/
```

```
class A;  
  A class B; /* B is a subclass of A */  
  ref (A) a;  
  ref (B) b;  
  proc assignA (ref (A) x)  
    begin  
      x := a  
    end;  
  assignA(b);
```

Smalltalk I

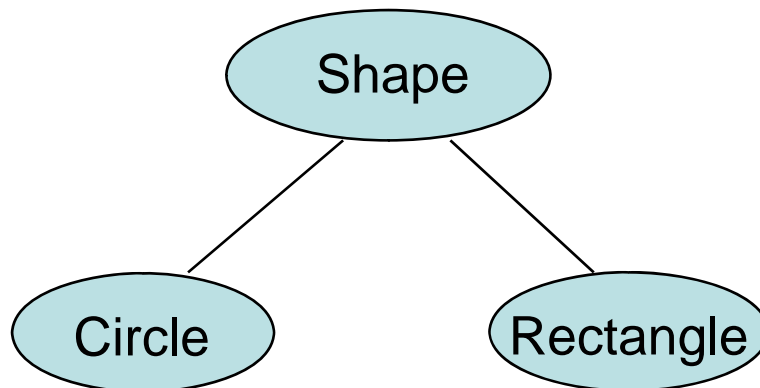




Example: Shapes

Interface of every *shape* must include *center*, *move*, *rotate*, *print*

- Programming Approach
 - General interface only in Shape
 - Different kinds of shapes are implemented differently
 - *Square*: four points, representing corners
 - *Circle*: center point and radius
- Modeling Approach
 - General interface *and* general implementation in shape
 - Shape has center point
 - Move moves by changing the position of the center point
 - 'To be or not be' virtual
 - e.g. Move should not be redefined in subclasses



In Simula, C++, a method specified as **virtual** may be overridden.

In Java, a method specified as **final** may *not* be overridden.

Subclass compatibility

- *Name compatible*: method with the same name in subclass overrides method in superclass (Smalltalk)
- *Structure compatible*: number and types of method parameters must be compatible (C++, Java)
- *Behavior compatible*: the effect of the subclass method must a specialization of the effect of the superclass method

Types versus classes

- Type as the set of operations on an object
 - Objects of two different classes may have the same type
 - Structural type compatibility
- Type = class (or interface)
 - Type name (e.g. class or interface name)

Example – Structural (sub) typing

- Smalltalk in C-syntax

```
class GraphicalObject {  
    move(dx,dy int) {...}  
    draw() {...}  
};  
...
```

```
...; r.draw(); ...; r.draw();
```


Example

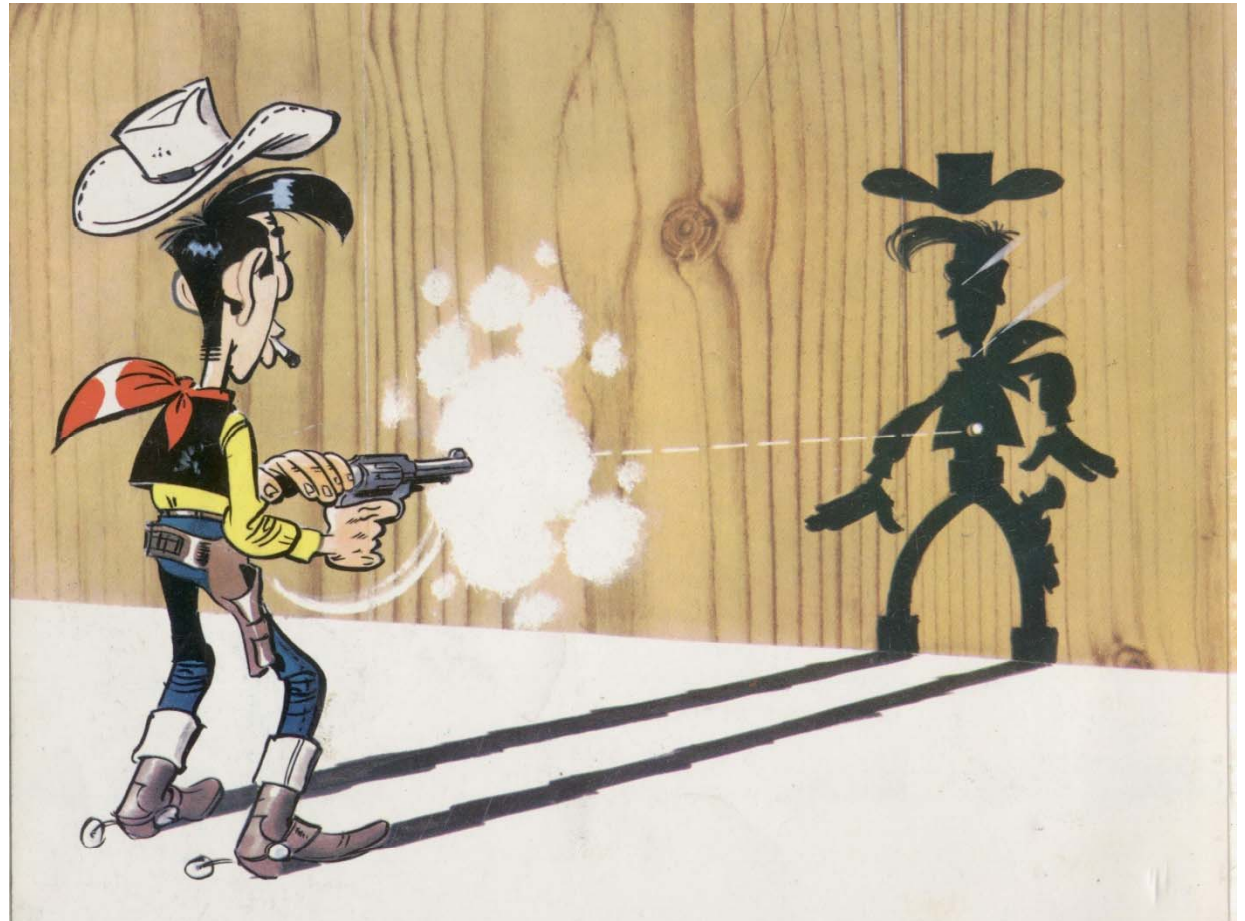
- Two classes with the same structural type

```
class GraphicalObject {  
  move(dx,dy int) {...}  
  draw() {...}  
};
```

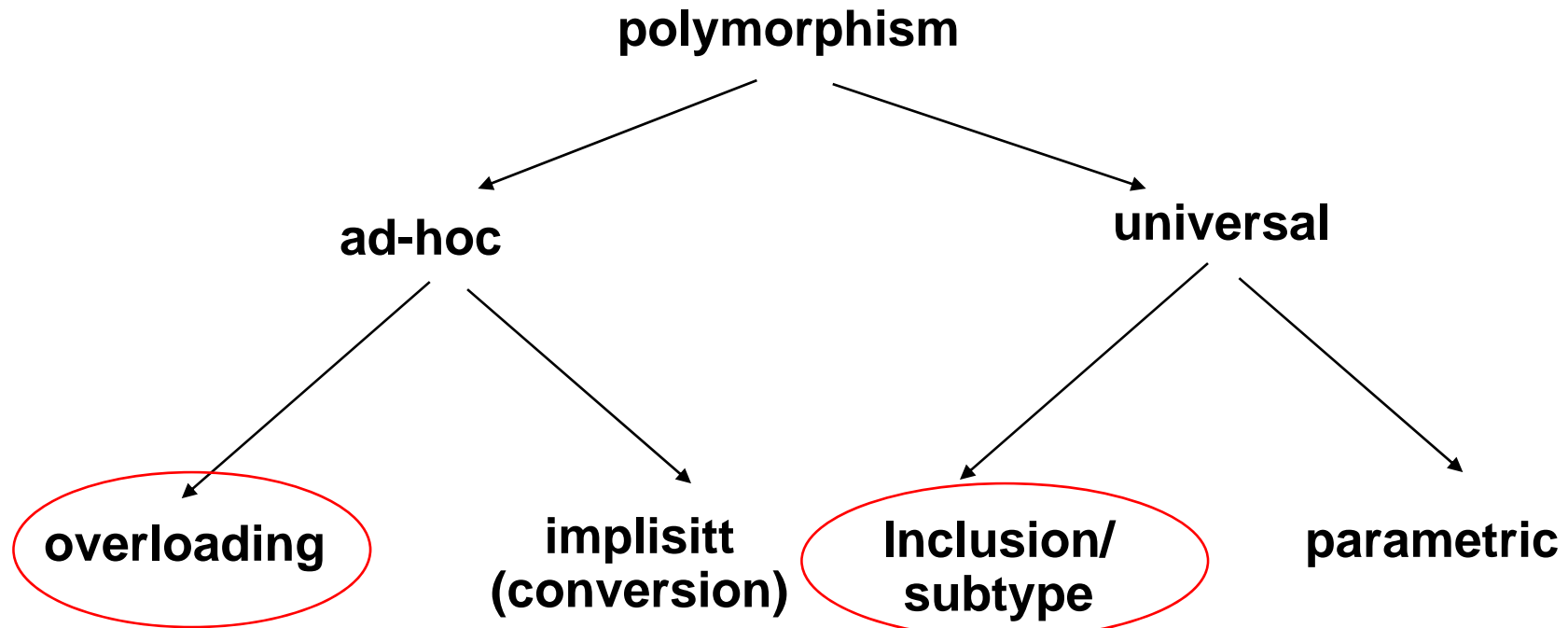
```
class Cowboy {  
  move(dx,dy int) {...}  
  draw() {...}  
};
```

...

```
...; r.draw(); ... ; r.draw();
```



Classification of polymorphism



Inclusion/subtype polymorphism

```
Point p1;  
ColorPoint cp1;
```

```
...; p1.equals(cp1); ...
```

'equals' works for cp1 because
ColorPoint is a subtype of type Point

```
class Shape {  
    void draw() {...}  
    ...  
};  
class Circle extends Shape {  
    void draw() {...}  
    ...  
};
```

```
...; aShape.draw(); ...
```

will draw a Circle if aShape is a Circle

Overriding vs Overloading – I

```
class Shape {
    ...
    bool contains(point pt) {...}
    ...
};

class Rectangle extends Shape {
    ...
    bool contains(int x,y) {...}
    ...
}
```

- Overloading
 - within the same scope {...},
 - crossing superclass boundaries

Overriding vs Overloading - II

```
class C {
    ...
    bool equals(C pC) {
        ...
    }
}

class SC extends C {
    ...
    bool equals(C pC) {
        ...
    }

    bool equals(SC pSC) {
        ...
    }
}
```

```
C c      = new C();
SC sc    = new SC();
C c'     = new SC();
```

```
c.equals(c)      //1
c.equals(c')     //2
c.equals(sc)     //3
```

```
c'.equals(c)     //4
c'.equals(c')    //5
c'.equals(sc)    //6
```

```
sc.equals(c)     //7
sc.equals(c')    //8
sc.equals(sc)    //9
```

Covariance/contravariance/novariance

```
class C {  
  T1 v;  
  T2 m(T3 p) {  
    ...  
  }  
}  
  
class SC extends C {  
  T1' v;  
  T2' m(T3' p){  
    ...  
  }  
}
```

- Covariance:
 - T1' subtype of T1
 - T2' subtype of T2
 - T3' subtype of T3
- Contravariance:
 - The opposite
- Novariance: same types
- Most languages have no-variance
- Some languages provide covariance on both: most intuitive
- Statically type-safe:
 - Contravariance on parameter types
 - Covariant on result type

Example: Point and ColorPoint – I: no variance

```
class Point {
  int x,y;
  move(int dx,dy) {
    x=x+dx; y=y+dy}

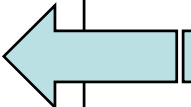
  bool equal(Point p) {
    return x=p.x and y=p.y
  }
}

class ColorPoint
  extends Point {
  Color c;
  bool equal(Point p) {
    return x=p.x and
           y=p.y and
           c=p.c
  }
}
```

```
Point p1, p2;
ColorPoint c1,c2;
```

1. p2.equal(p1)
2. c2.equal(c1)
3. p1.equal(c1)
4. c1.equal(p1)

```
return super.equal(p) and
       c=p.c
```



Example: Point and ColorPoint – II: covariance

```
class Point {
  int x,y;
  move(int dx,dy) {
    x=x+dx; y=y+dy}

  bool equal(Point p) {
    return x=p.x and y=p.y
  }
}

class ColorPoint
  extends Point {
  Color c;
  bool equal(ColorPoint cp) {
    return super.equal(cp) and
      c=cp.c
  }
}
```

```
Point p1, p2;
ColorPoint c1,c2;
```

1. p2.equal(p1) OK run-time
2. c2.equal(c1) OK run-time
3. p1.equal(c1) OK run-time
4. c1.equal(p1) NOK run-time

Example: Point and ColorPoint – III: casting

```
class Point {
  int x,y;
  move(int dx,dy) {
    x=x+dx; y=y+dy}

  bool equal(Point p) {
    return x=p.x and y=p.y
  }
}

class ColorPoint
  extends Point {
  Color c;
  bool equal(Point p) {
    return super.equal(p) and
      c=(ColorPoint)p.c
  }
}
```

```
Point p1, p2;
ColorPoint c1,c2;
```

1. p2.equal(p1)
2. c2.equal(c1)
3. p1.equal(c1)
4. c1.equal(p1)

Example: Point and ColorPoint –

```
class Point {
  int x,y;
  virtual class ThisClass < Point;

  bool equal(ThisClass p) {
    return x=p.x and y=p.y
  }
}

class ColorPoint
  extends Point {
  Color c;
  ThisClass:: ColorPoint;
  bool equal(ThisClass p) {
    return super.equal(p) and
           c=p.c
  }
}
```

- Alternative to casting:
 - Virtual classes with constraints (OOPSLA '89)
 - Still run time type checking

Compile-time vs run-time type checking

- $f(x)$
 - if $f : A \rightarrow B$ then $x : A$, but when to check?
- Compile-time (static)
 - Must ensure that x will never have another type than A
- Run-time (dynamic) type checking
 - When calling $f(x)$ – determine the type of x and check if it is A
- Basic tradeoff
 - Both prevent type errors
 - Run-time checking slows down execution
 - Compile-time checking restricts program flexibility
- Combined compile - and run-time checking

Two questions

1. How is the type of an entity (variable, function, ...) specified?
2. When is the type determined?

2 \ 1	Explicit declaration	Implicit declaration
Static (compile-time)	Java, C++, Algol, Simula ML, BETA	ML, Perl
Dynamic (run-time)	Simula, BETA, Java	Smalltalk