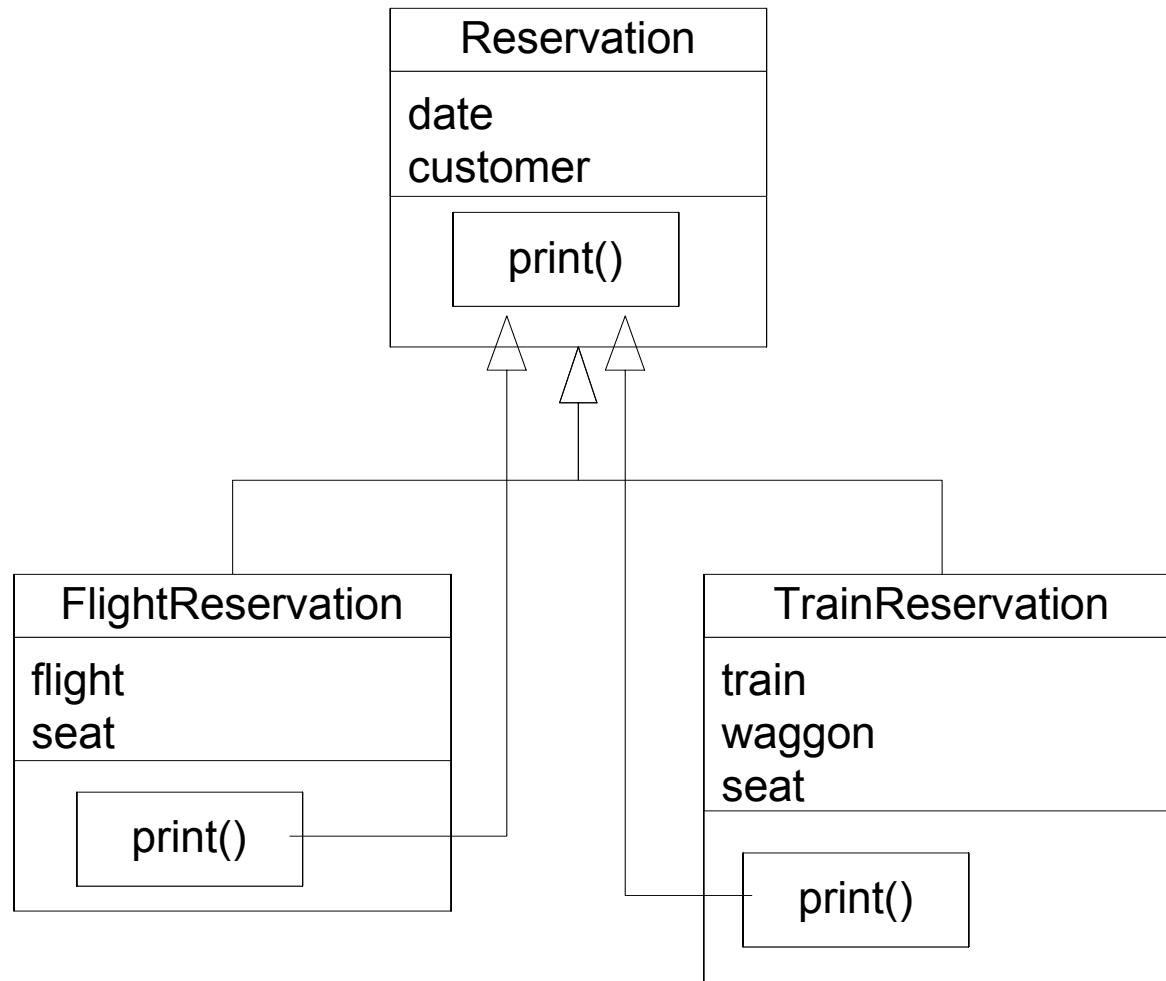


'Subtyping' for behaviour?



'Subtyping' for behaviour – BETA style

```
class Reservation {  
    date . . .; customer . . .;  
    void print() {  
        // print Date and Customer  
        inner;  
    }  
}
```

```
class FlightReservation  
    extends Reservation {  
    flight . . .; seat . . .;  
    void print extended {  
        // print flight and seat  
        inner;  
    }  
}
```

'Subtyping' for behaviour – Java style

```
class Reservation {  
    date . . .; customer . . .;  
    void print() {  
        // print date and Customer  
    }  
}  
  
class FlightReservation  
extends Reservation {  
    flight. . .; seat. . .;  
    void print {  
        super.print();  
        // print Flight and Seat  
    }  
}
```

- super.print() ==
(Reservation)this.print()
- Does the BETA solution give
'behavioral compatibility'?
- Is it possible to have both super
and inner? OOPSLA 2004
- What if we turn print into a static
method?

```
class Point { int x, y; }
```

is equivalent to the declaration:

```
class Point { int x, y;  
    public Point() { super(); }  
}
```

Subtyping

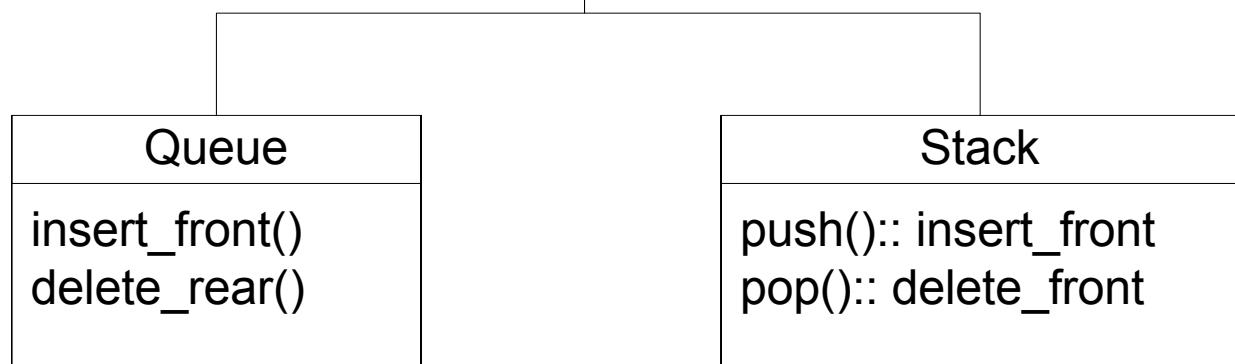
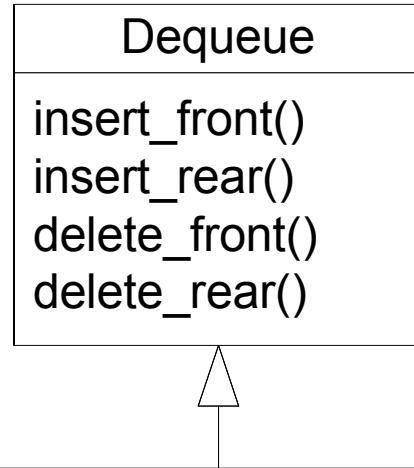
=

subclassing??

Queue
insert()
delete()

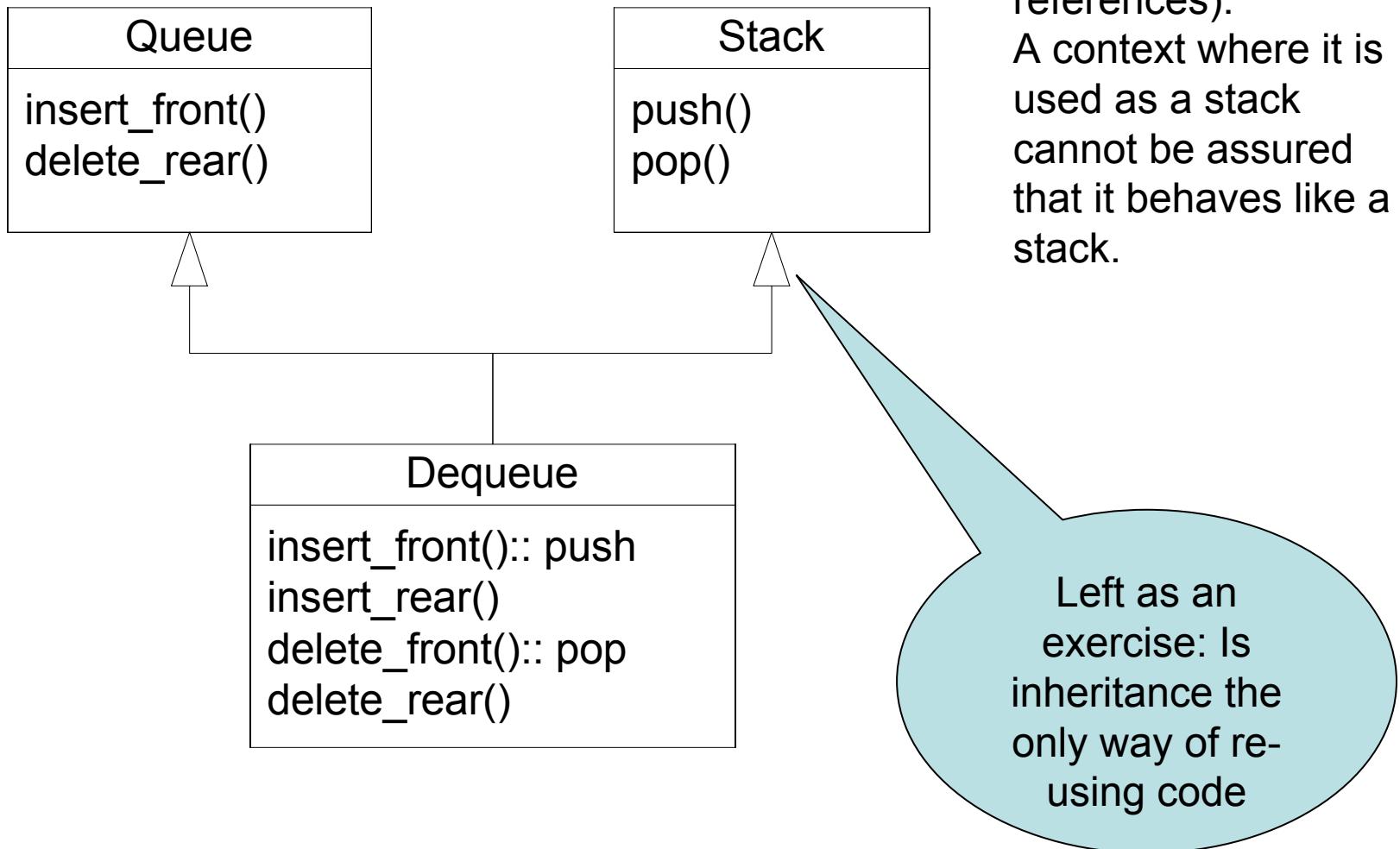
Stack
push()
pop()

Dequeue
insert_front()
insert_rear()
delete_front()
delete_rear()



```
Dequeue d; Stack s; Element e;
void f(Dequeue dp, Element ep) {
    dp.insert_front(ep); dp.insert_rear(ep) }
...
f(s, e)
```

The opposite any better?

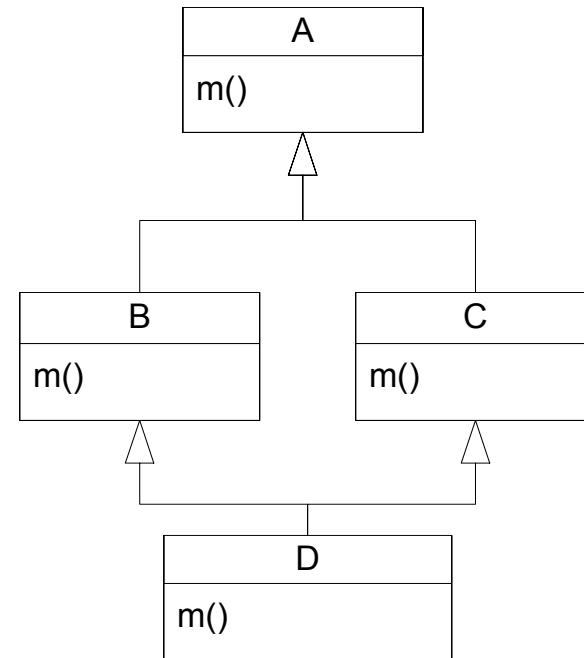


Subtyping = subclassing (Smalltalk)

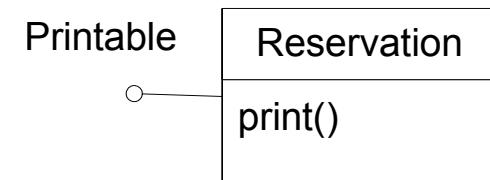
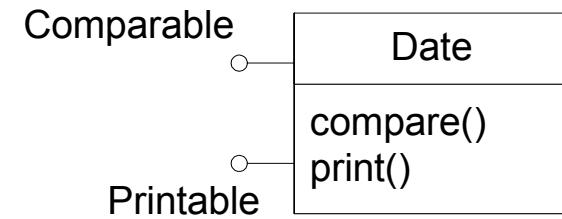
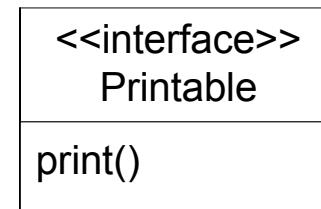
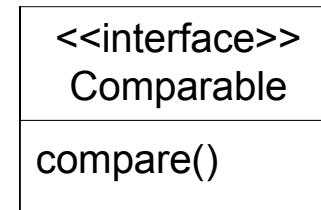
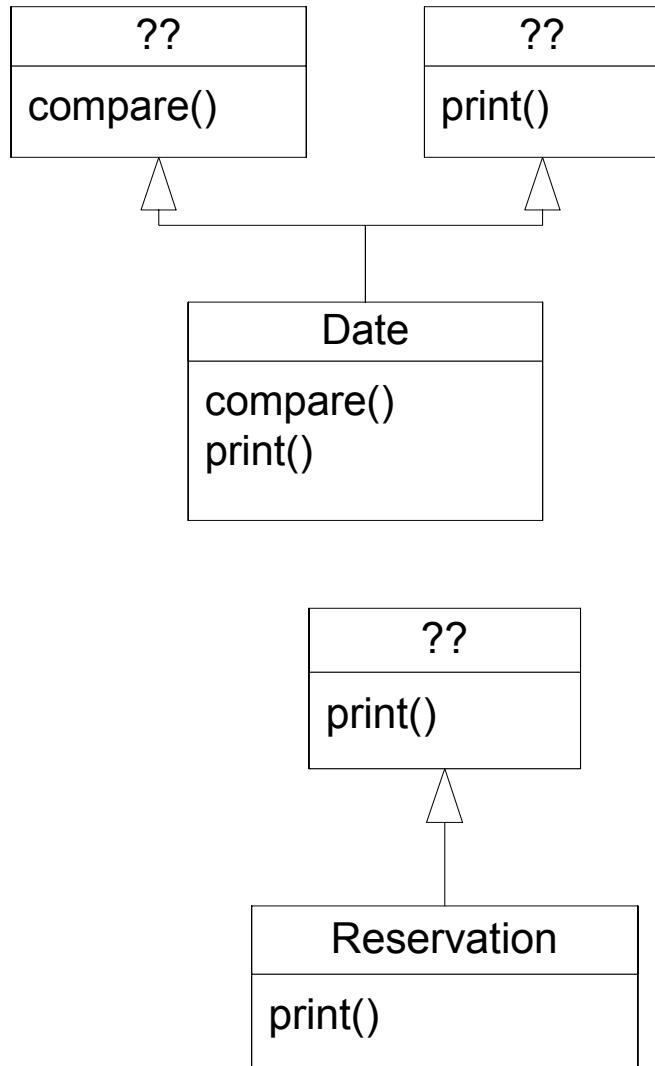
- Mitchell 11.7
- Smalltalk untyped, so how?
 - Subtyping as a relation between interfaces, substitutability
- Class Set
 - Set_interface ={isEmpty, size, includes, add}
- Class ExtensibleCollection
 - Set_interface ={isEmpty, size, includes, add}
- An ExtensibleCollection object can take the place of a Set object
 - There will be no 'message not understood'
- Remember the cowboy ...; r.draw();,

Multiple inheritance I

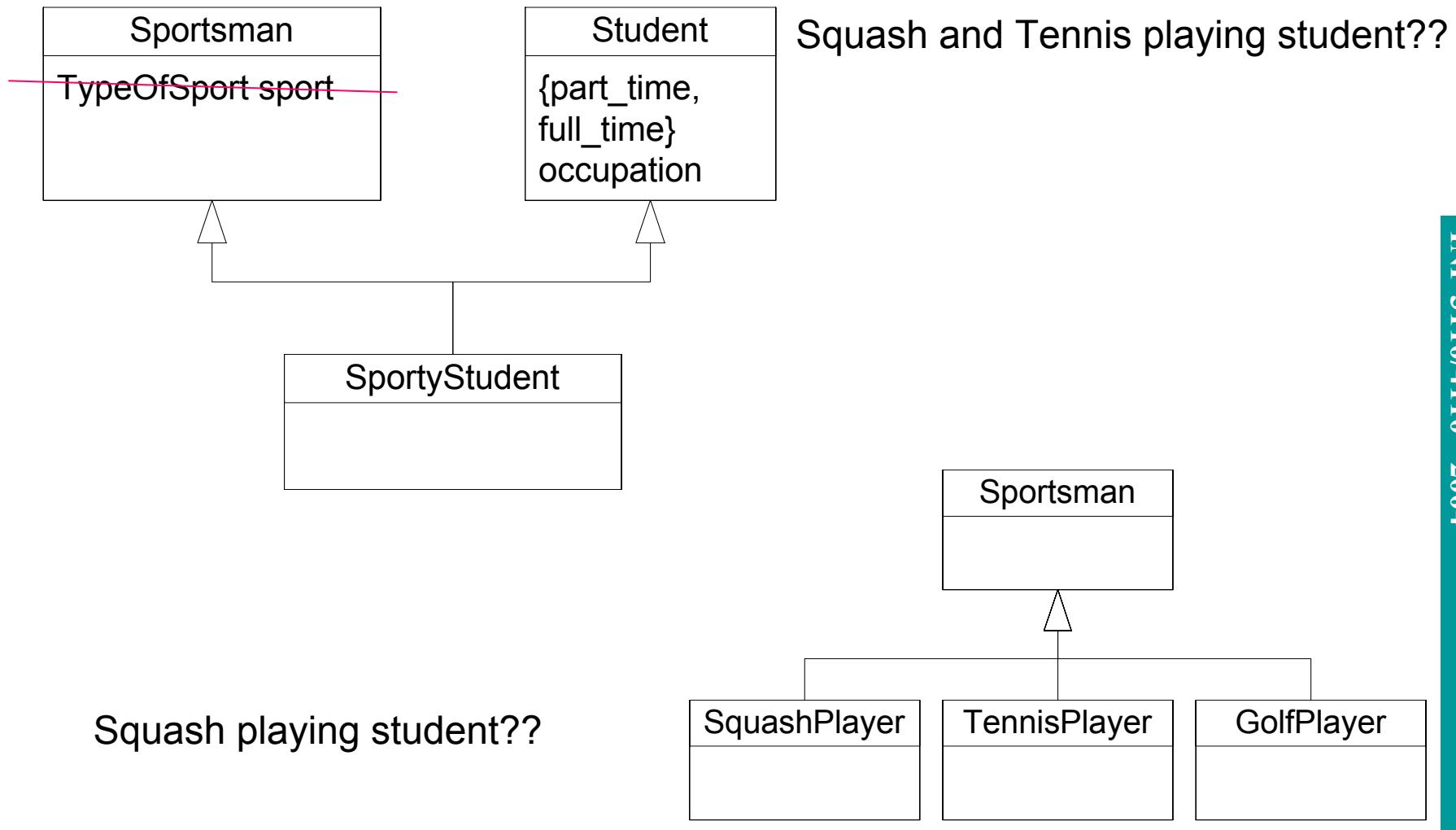
- Multiple supertypes or just multiple implementations?
- Name conflicts - `m()`
 - Take the leftmost (i.e. '`B.m()`')
 - Not allowed
 - Renaming
 - Explicit identification '`B.m()`'
 - In definition of class D
 - In every use of `m()`
- Overriding
- One or two A's?



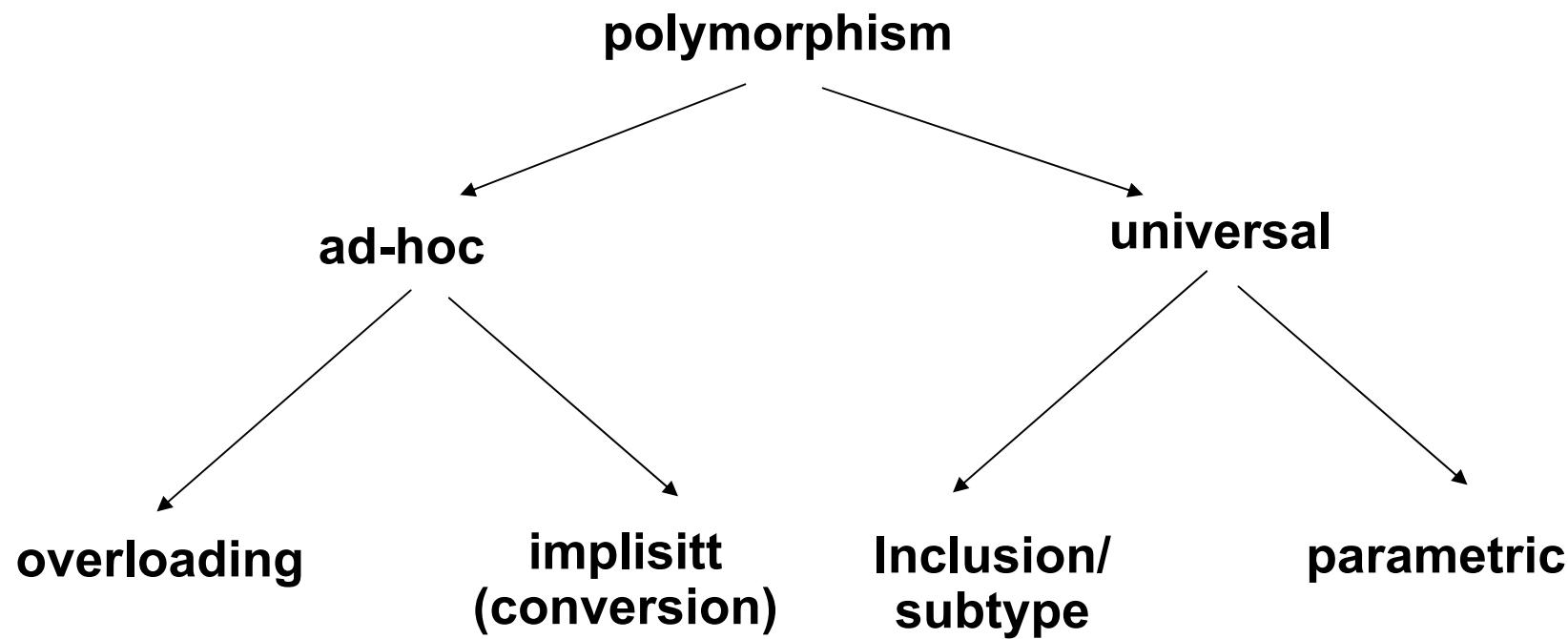
Multiple inheritance II



Multiple classification



Classification of polymorphism



Inclusion/subtype polymorphism

```
Point p1;  
ColorPoint c1;
```

```
...; p1.equal(cp1); ...      'equal' 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 y,y) {...}  
    ...  
}
```

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

Overriding vs Overloading - II

```
class C {  
    ...  
    bool equals(C pC) {  
        ...  
        // equals 1  
    }  
}  
  
class SC extends C {  
    ...  
    bool equals(C pC) {  
        ...  
        // equals 1  
    }  
  
    bool equals(SC pSC) {  
        ...  
        // equals 2  
    }  
}
```

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 novariance
- 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  
    }  
}
```

- Towards a solution
- Virtual classes with constraints
(OOPSLA '89)

Multi-methods, 'dynamic overloading'

```
class Point { int x,y; }

class ColorPoint extends Point { Color c; }

bool equal(Point p1, Point p2) {
    return p1.x=p2.x and p1.y=p2.y };

bool equal(ColorPoint p1, ColorPoint p2) {
    return p1.x=p2.x and p1.y=p2.y and p1.c=p2.c };

equal (pt1, pt2);

    - equal (aPoint, aPoint);
    - equal (aPoint, aColorPoint);
    - equal (aColorPoint, aColorPoint);
```

Java generics

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer)myIntList.iterator().next()
```

```
List<Integer>myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next()
```

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}  
  
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}  
  
public interface IntegerList {  
    void add(Integer x)  
    Iterator<Integer> iterator();  
}
```

Generics and subtyping

```
List<String> ls = new ArrayList<String>(); //1  
  
List<Object> lo = ls; //2  
  
lo.add(new Object()); // 3  
  
String s = ls.get(0); // 4: attempts to assign an  
                      Object to a String!
```

Bounded polymorphism - Wildcards - I

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) { System.out.println(i.next());}  
}
```

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) { System.out.println(e); }  
}
```

```
void printCollection(Collection<?> c) {  
    for (Object e : c) { System.out.println(e);}  
}
```

```
Collection<?> c = new ArrayList<String>();  
c.add(new Object()); // compile time error
```

Bounded polymorphism - Wildcards - II

```
public abstract class Shape {  
    public abstract void draw(Canvas c);  
}  
public class Circle extends Shape {  
    private int x, y, radius;  
    public void draw(Canvas c) { ... }  
}  
public class Rectangle extends Shape {  
    private int x, y, width, height;  
    public void draw(Canvas c) { ... }  
}  
  
public class Canvas {  
    public void draw(Shape s) { s.draw(this);}  
}
```

Bounded polymorphism - Wildcards - III

```
public void drawAll(List<Shape> shapes) {  
    for (Shape s: shapes) { s.draw(this);}  
}
```

```
public void drawAll(List<? extends Shape> shapes) { ... }
```

Generic methods

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {  
    for (Object o: a) {  
        c.add(o); }          // compile time error  
}
```

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {  
    for (T o: a) {  
        c.add(o); }  
}
```

```
class Collections {  
    public static <T> void copy(List<T> dest, List<? extends T> src) {...}  
}
```

```
class Collections {  
    public static <T, S extends T> void copy(List<T> dest, List<S> src) {...}  
}
```

Modularity - Chapter 9 : Basic Concepts

- Component
 - Meaningful program unit
 - Function, data structure, module, ...
- Interface
 - Types and operations defined within a component that are visible outside the component
- Specification
 - Intended behavior of component, expressed as property observable through interface
- Implementation
 - Data structures and functions inside component
 - Representation independence

Example: Function Component

- Component
 - Function to compute square root
- Interface
 - float sqroot (float x)
- Specification
 - If $x > 1$, then $\text{sqrt}(x) * \text{sqrt}(x) \approx x$.
- Implementation

```
float sqroot (float x){  
    float y = x/2; float step=x/4; int i;  
    for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step = step/2;}  
    return y;  
}
```

'programming-in-the-small' versus 'programming-in-the large'

Module language concept

```
module Set
  interface
    type set
    val empty : set
    fun insert : elt * set -> set
    fun union : set * set -> set
    fun isMember : elt * set -> bool
  implementation
    type set = elt list
    val empty = nil
    fun insert(x, elts) = ...
    fun union(...) = ...
    ...
end Set
```

- Can define ADT
 - Private type
 - Public operations
- More general
 - Several related types and operations
- Some languages
 - Separate interface and implementation
 - One interface can have multiple implementations

Modules in object oriented languages

- Classes?
 - Interfaces
 - Types?
 - Operations?
 - Implementation
 - Representation independence?
 - State?
- Packages?
 - Interfaces?
 - Types?
 - Operations?
 - Implementation
 - State?
- EJB/.NET Components?
 - Yes
 - Inner classes?
 - Methods
 - Depends: Interface or implementation inheritance
 - Yes
 - No, but
 - Public classes
 - Static methods
 - No
 - Special-made classes

Encapsulation versus composition

```
class Apartment {  
    Kitchen theKitchen = new Kitchen();  
    Bathroom theBathroom = new Bathroom();  
    Bedroom theBedroom = new Bedroom ();  
    FamilyRoom theFamilyRoom = new FamilyRoom ();  
    ...  
    Person Owner;  
    Address theAddress = new Address()  
};  
  
...; myApartment.theKitchen.paint(); ...
```

```
class Point {  
    int x,y;  
    Point(int i,j) {  
    }  
};
```

Inner classes - locally defined classes

```
class Apartment {  
    Height height;  
    Kitchen theKitchen = new Kitchen {... height ...}();  
    class ApartmentBathroom extends Bathroom {... height ...}  
        ApartmentBathroom Bathroom_1 = new ApartmentBathroom ();  
        ApartmentBathroom Bathroom_2 = new ApartmentBathroom ();  
    Bedroom theBedroom = new Bedroom ();  
    FamilyRoom theFamilyRoom = new FamilyRoom ();  
    ...  
    Person Owner;  
    Address theAddress = new Address()  
};
```