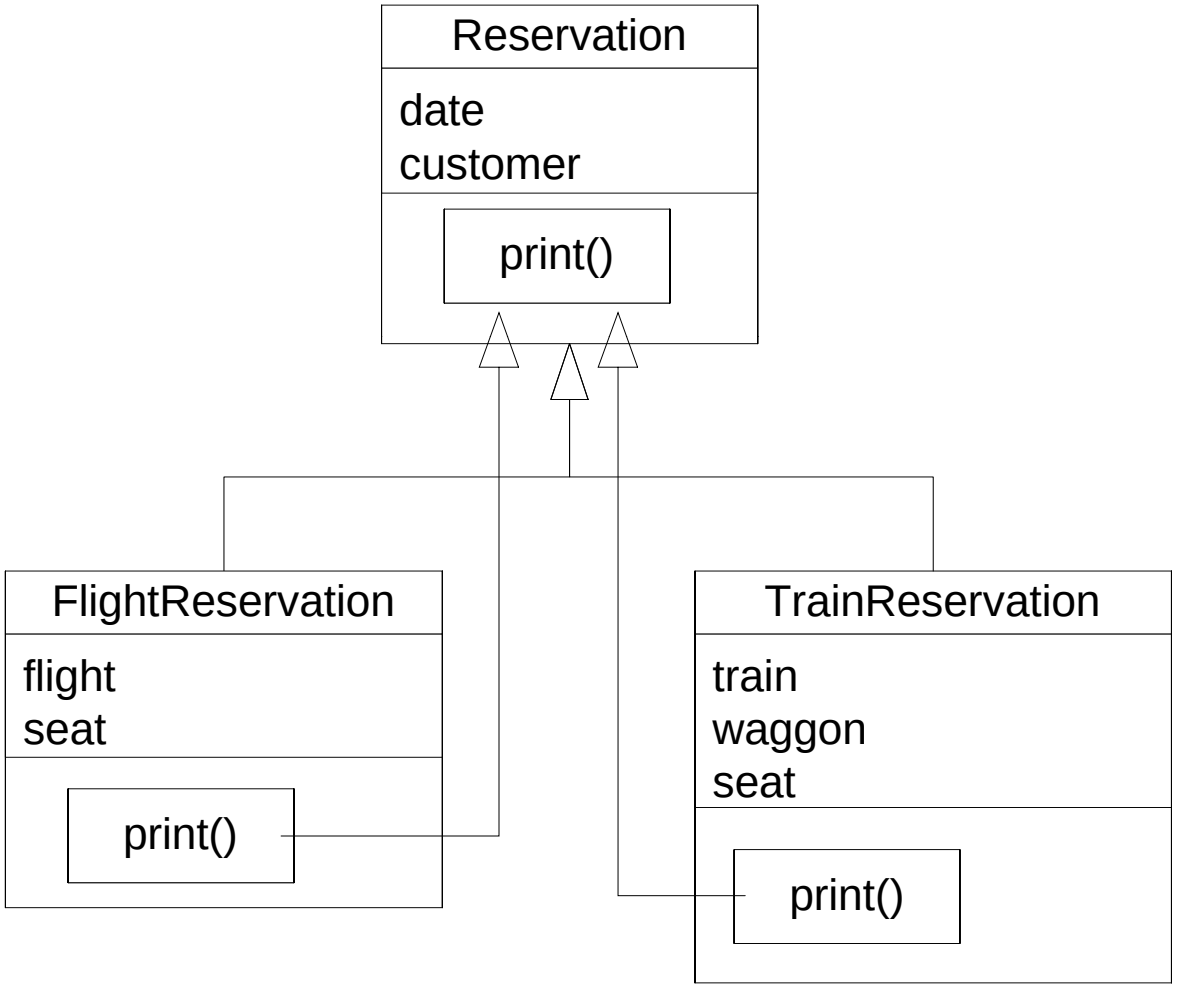
’Subtyping’ for behaviour?



’Subtyping’ for behaviour – the inner 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 – the super 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 inner style give ’behavioral compatibility’?
- What if we turn print into a static method?

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

is equivalent to the declaration:

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

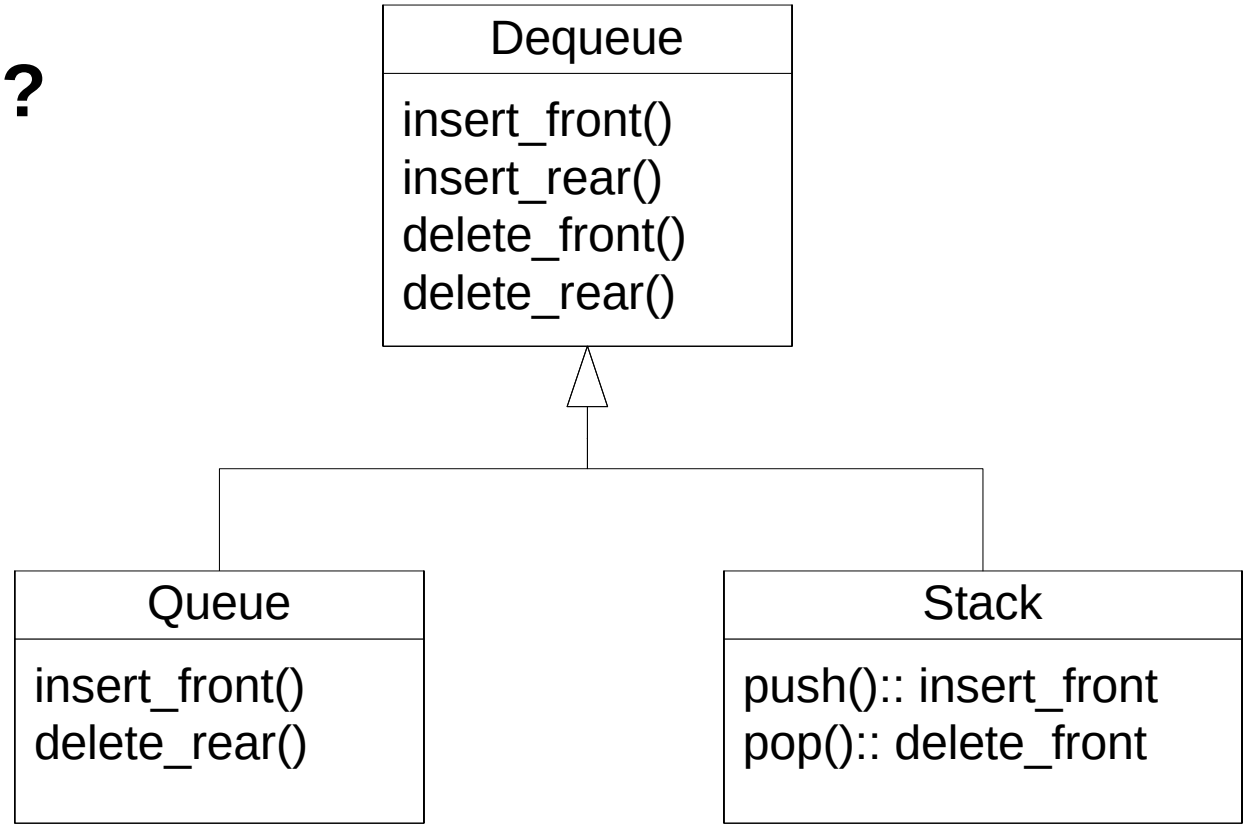
INF 3110 - 2008

Subtyping = subclassing??

Queue
insert() delete()

Stack
push() pop()

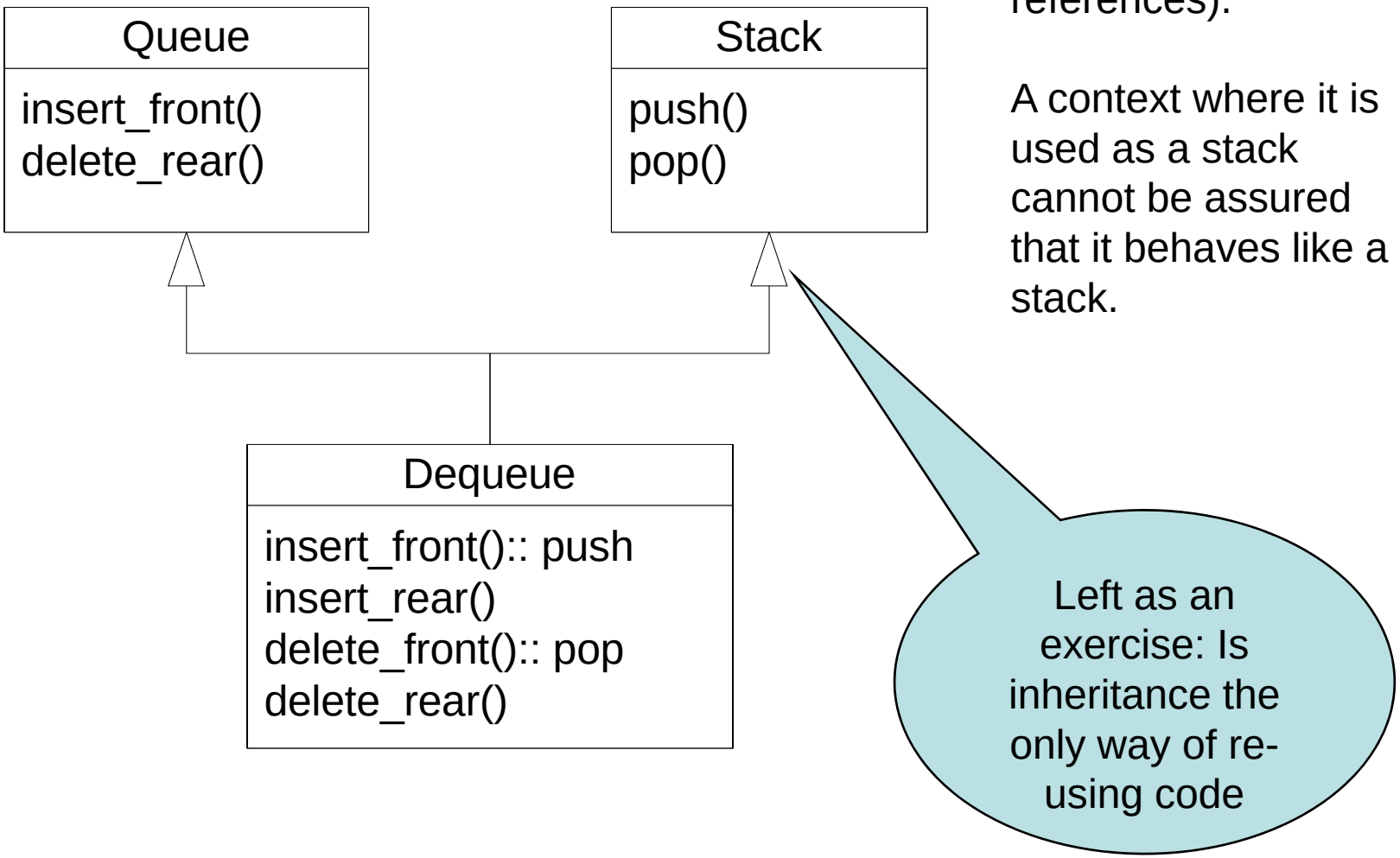
Dequeue
insert_front() insert_rear() delete_front() delete_rear()




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

INF 3110 - 2008

The opposite any better?



Can be substituted for both a Queue and a Stack (via different references).

A context where it is used as a stack cannot be assured that it behaves like a stack.

Subtyping = subclassing (Smalltalk)

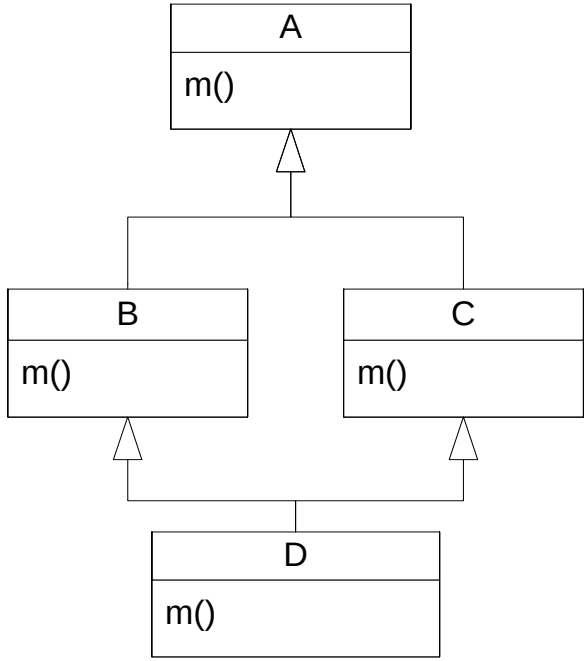
- Mitchell 11.7
- Smalltalk untyped, so how?
  - Subtyping as a relation between interfaces, substitutability
- Class Set
  - Set_interface ={isEmpty, size, includes, add}
- Class ExtensibleCollection
  - Set_interface ={isEmpty, size, includes, add}

- An ExtensibleCollection object can take the place of a Set object
  - There will be no ’message not understood’

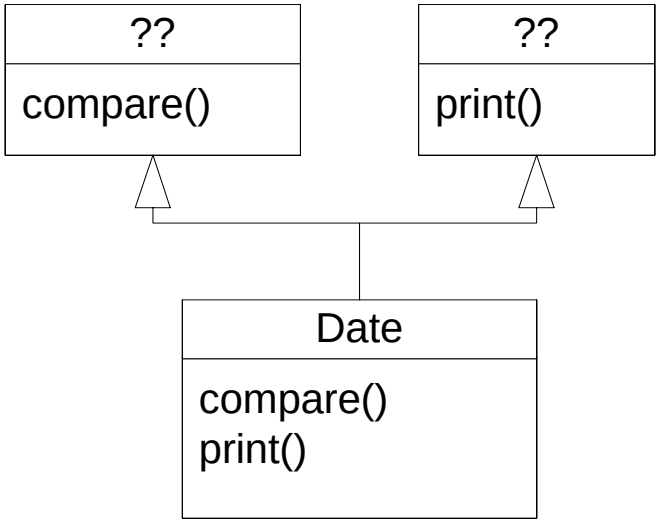
- Remember the cowboy ...; r.draw(); ...,

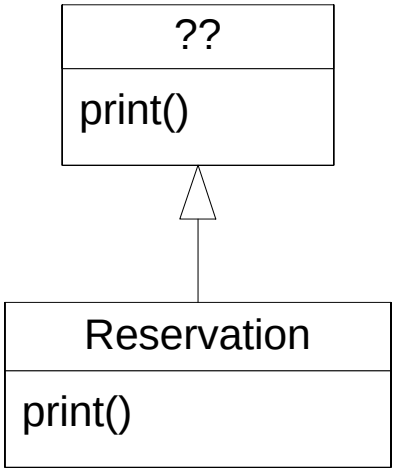
Multiple inheritance I

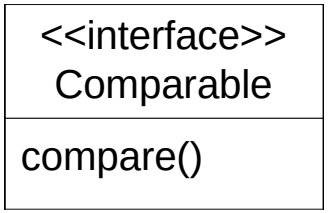
- Multiple supertypes or just multiple implementations?
- Name conflicts - m()
  - Take the leftmost (i.e. 'B.m()')
  - Not allowed
  - Renaming
  - Explicit identification 'B.m()'
    - In definition of class D
    - In every use of m()
- One or two A's?
- Overriding

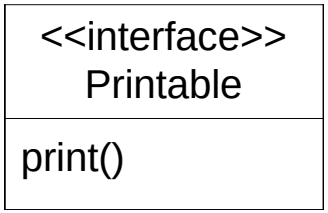


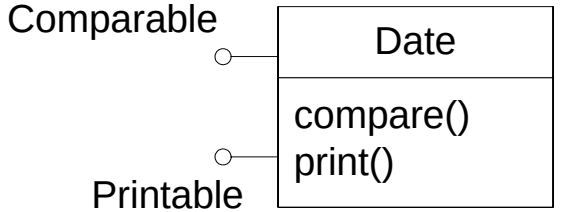
Multiple inheritance II

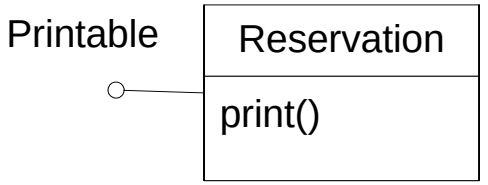






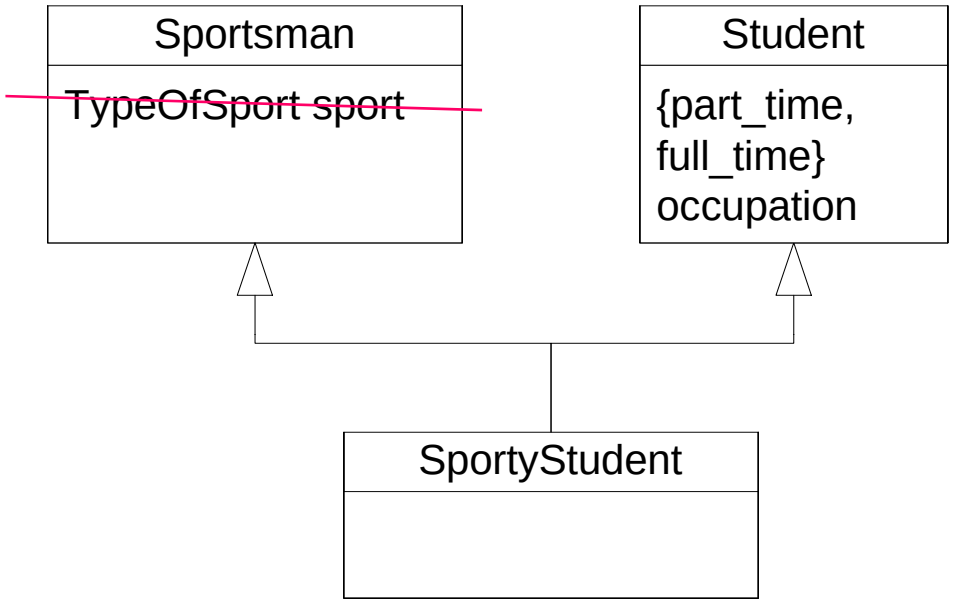






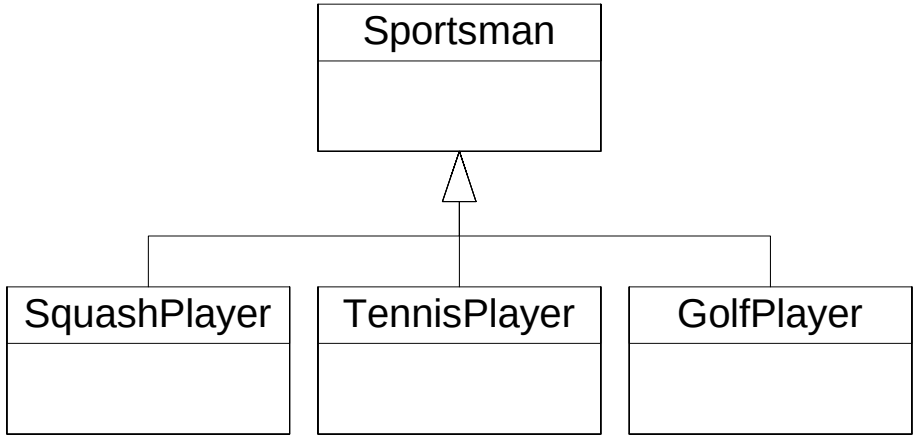
INF 3110 - 2008

Multiple classification



Squash and Tennis playing student??

Squash playing student??



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);

Constraining type parameters

- C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A[count] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- What parts of implementation depend on type?

Meaning and implementation of <

Java generics

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

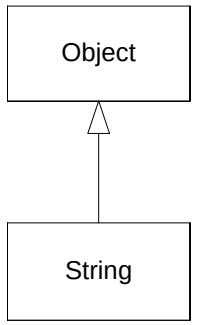
```
List<Integer>myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next()
```

```
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

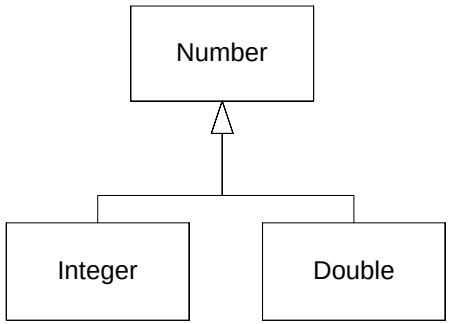
- String subclass of Object ~~=>~~ List<String> subclass of List<Object>

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



- Integer subclass of Number ~~=>~~ List<Integer> subclass of List<Number>

```
List<Integer> ints = Arrays.asList(1,2);
List<Number> nums = ints;  // compile-time error
nums.add(3.14);
```



INF 3110 - 2008

Unbounded polymorhpism - 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<any type> **not** subtype of Collection<Object>
- Collection<any type> subtype of Collection<?>

Bounded polymorhpism - 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 polymorhpism - Wildcards - III

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

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

- List<S> subtype of List<? extends Shape > for every S being a subtype of Shape
- List<S> subtype of List<? extends T > for every S being a subtype of T

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) {…}
}
```

```
interface Sink<T> { flush(T t);
}

public static <T> T writeAll(Collection<T> coll, Sink<T> snk){
  T last;
  for (T t : coll) {
    last = t; snk.flush(last);
   }
   return last;
}
...
Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // illegal call
```

```
public static <... T> T writeAll(Collection<? extends T>, Sink<T>){...}

String str = writeAll(cs, s); // call ok, but wrong return type
```

```
public static <T> T writeAll(Collection<T> coll, Sink<? super T> snk){...}

String str = writeAll(cs, s); // Yes!
```

Generic classes in Java (and C#)

```
interface I{…}
class C{…}
```

- Generic class

```
class G<T extends C implements I> {
   … T is known as a C and an I …  }
```

```
class G<T> where T:C, I {
  … T is known as a C and an I …   }
```

- Actual parameter

```
class AC extends C implements I{ … }
```

- Used like this:

```
G<AC> g = new G<AC>( );
```

- Java
  - Generic parameters can be classes and interfaces
- C#
  - Generic parameters can be classes and interfaces, and predefined types

Java

- Implementation
  - Type parameters removed (erased), substituted by `Object`
  - Castings inserted according to the actual parameters
- Restrictions
  - Can *not* cast to `G<A>` or use `instanceof G<A>`
  - Can *not* cast to `T` or use `instanceof T`
  - `new T( )` *not* allowed
  - All classes `G<A>`, where `A` is an actual parameter have a common set of static variables and methods; therefore:
    - Types of these can not depend on `T`
  - `A1` subtype of `A2` does *not* imply `G<A1>` subtype of `G<A2>`

C#

- Implementation
  - Makes a runtime descriptor for all actual uses of a generic class.
  - Uses the same code for all `G<A>`s as far as possible (i.e. as long as acatual parameters are classes and interfaces)
- Fewer restrictions
  - casting and `instanceof` are allowed, both with `T` (within `G`) and with `G<A>`.
  - `new T( )` allowed if `T` specified like this:

```
class G<T> where T: C, new( ) { … }
```

- Naming classes with actual parameters:

```
using GA = G<A>;
```

Java and C#

- T can *not* be super class of an inner class.
- F-bounded polymorphism

```
class G<T extends G<T>>
```

  - Actual parameter for T is e.g.:

```
class A extends G<A>
```

- Generic methods (including static methods):
  - Java: `<U extends B>U getValue  (int i, U u) {…}`
  - C#: `U getValue <U> (int i, U u) where U: B {…}`

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 sqrt(x)*sqrt(x) ≈ 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?
  - Interface
    - Types?
    - Operations? Functions?
  - Implementation
    - Representation independence?
    - State?
- Packages?
  - Interface
    - Types?
    - Operations? Functions?
  - Implementation
    - State?
- EJB/.NET Components?
-
  - Yes, but
    - Interfaces/Inner classes?
    - Methods/Static methods
  -
    - Depends: Interface or implementation inheritance
    - Yes
-
  - No, but
    - Public interfaces/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, int 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()
}
```