Why are statically-typed OOL's so inflexible?

Java programs require lots of type casts (as do C++, Object Pascal, etc.).
Why?

The Object class in Java illustrates most of the problems.

```
public class Object{
   protected Object clone(){…}
   public boolean equal(Object other){…}
}
```

Recall not allowed to change signature of methods in extensions.

Also all classes automatically inherit from Object.

```
public class A implements Cloneable{
   protected B b = …;
   …
   public Object clone(){
      A other = super.clone();
      other.b = b.clone();               // type error
      return other;
   }
}

A a1 = new A();
A a2 = a1.clone();                   // type error
```

Both errors would disappear if could change
return types of methods. (OK in C++)

Suppose also want to override equals in A:

```
   public boolean equal(Object other){
      if (other instanceof A){
```

```
    A aother = (A)other;
    return (b = aother.b);
  else
    ???
```

Problems:

- Inconvenient (and slow) to have run-time test
  No static check possible.
- What to do in else clause?
  If raise exception must declare in method header.

Get similar problems if try to define doubly-linked node as subclass of linked node.

```
public void setNext(Node newNext){…}
```

Want parameter type to be doubly-linked in subclass.

Methods where parameters should be of same type (class) as receiver called "binary methods".

Source of many typing problems in OOL's.

Still other problems:

```
public class Circle{
  protected Point center;
  …
}
```

If define ColorCircle as extension, might want center to be ColorPoint.

Can't make any of these changes in signature (types) in Java, Object Pascal, and C++
(aside from return type in C++)

Why are there these restrictions?

Is there any way to overcome them?

Look at the following example:

```
class A{
   D m(C c){…}
   void n(){…  self.m(someC) …}
}

class B{
   D' m(C' c'){…}
}
```

For which C', D' will B end up being type safe if A is?

Homework asks similar question for instance variables.

GJ: Adding F-bounded polymorphism to Java

Odersky, Wadler, et al (follow up to Pizza)

GJ adds parametric polymorphism w/ syntax like C++'s templates:

```
   public class Stack<Elt> extends Vector<Elt>{
       public Elt push(Elt item){...}
       public Elt pop(){...}
       public Elt peek(){...}
       public boolean empty(){...}
       public int search(Elt o){...}
    }

   Stack<Point> myStack = new Stack<Point>();
  Point aPoint = new Point(2,3);
  myStack.push(aPoint);
```

Can also add constraints to type parameters:

```
public interface Comparing {
  public boolean equal(Comparing other);
  public boolean greaterThan(Comparing other);
  public boolean lessThan(Comparing other);
}

public class OrderedList<Elt implements Comparing>
                                extends ... {
   protected Elt[] elts = new Elt[0];

  public void insert(Elt item){…
     while (elts[current].greaterThan(item))
        current ++;
          …
     }
   }

   public Elt removeFirst(){...}
  public boolean empty(){...}
  public int searchFor(Elt o){...}
}
```

How to define ordered objects?

```
public class KeyedObj implements Comparing{
   protected int key ;
   ...;
   public int getKey(){…}
   public lessThan(Comparing other){
     return this.key < other.getKey();}
}
```

Won't work: other.getKey() not well-typed!

```
   public lessThan(Comparing other){
```

```
    if (other instanceof Comparing)
      return this.key < other.getKey();}
    else
      ?????
```

Same problem as earlier!

F-bounded polymorphism (1989) can help:

```
public interface Comparing <Elt>{
  boolean lessThan(Elt other);
  boolean greaterThan(Elt other);
}

public class OrderedList
               < Elt implements Comparing<Elt> >{
  protected Elt [] elts;
  public void insert(Elt newVal){
    while (elts[current].greaterThan(newVal))
       current ++;
       …
  }
}

public class KeyedObj
                implements Comparing <KeyedObj>{
  protected int key ;
  ...;
  public int getKey(){…}
  public lessThan(KeyedObj other){
    return this.key < other.getKey();}
}
```

Now OrderedList<KeyedObj> is fine!

Generally works well (though confusing at first).
Still one problem -- F-bounded not preserved under subclass:

```
public class NuKeyedObj extends KeyedObj {
  protected String nuField;
    ...;
  public lessThan(KeyedObj other){
    return this.key < other.getKey() &&
                      other.getNuField() ...;}
}
```

Unfortunately, NuKeyedObj does not implement Comparing <NuKeyedObj>.

Can't be used with OrderedList!

Other info on GJ:

- Works with existing JVM -- essentially translates to original code w/Object and casts.
- Authors designed so that existing library classes can be used as though they were polymorphic.
- Because of translation, cannot get accurate info using Java's reflection facilities or debugger.
- See GJ web page available through hmwk page.

Eiffel

Designed by Bertrand Meyer in mid-80's

Class-based OOL w/multiple inheritance

Assertions: pre- and post-conditions, loop invariants and variants built into language.

Supports bounded polymorphism.

Reference semantics like Java, garbage collection, etc.

Information hiding: private, public, or could list classes visible to (like C++'s friends)

No interfaces or modules.

In subclasses, can redefine or even rename methods.

Can also change type of instance variables, parameters and return types covariantly.
Seen this can cause type-safety problems!

Introduced "anchor" types:
Can declare type to be "like" another feature:

x: A;
y: like x;

Current is Eiffel's name for self.

Example:

```
class LINKABLE [G]

feature

   item: G;                -- value held
   right: like Current;    -- Right neighbor

   putRight (other: like Current) is
        -- Put `other' to right of current cell.
      do
         right := other
      ensure
         chained: right = other
      end;

end -- class LINKABLE
```

```
class BILINKABLE [G] inherit

        LINKABLE [G]
           redefine
              putRight
           end

feature -- Access

   left: like Current;   -- Left neighbor

   putRight (other: like Current) is
         -- Put `other' to right of current cell.
      do
         right := other;
         if (other /= Void) then
            other.simplePutLeft (Current)
         end
       end;

    putLeft (other: like Current) is
           -- Put `other' to left of current cell.
        do
           left := other;
           if (other /= Void) then
              other.simplePutRight (Current)
           end
        ensure
           chained: left = other
        end;

feature {BILINKABLE}

   simplePutRight (other: like Current) is
         -- set `right' to `other'
```

```
      do
         right := other
      end;

   simplePutLeft (other: like Current) is
         -- set `left' to `other'
      do
         left := other
      end;

invariant

   rightSymmetry:
      (right /= Void) implies (right.left = Current);
   leftSymmetry:
      (left /= Void) implies (left.right = Current)

end -- class BILINKABLE
```

Notice BILINKABLE is subclass of LINKABLE.
Can't do this with Java, C++, etc.
Secret is use of "like Current" as type of instance variables and in types of methods.

Can define:

```
class LINKEDLIST[NODE -> LINKABLE] …
```

Can be instantiated with either

- LINKABLE (and get singly-linked list) or
- BILINKABLE (and get doubly-linked list).

Very expressive w/out F-bounded polymorphism:

```
deferred class Comparing
feature
   lessThan(other: like Current): boolean
                                         is deferred
   end
```

```
   greaterThan(other: like Current): boolean;
end
```

Unfortunately, use of like Current gives rise to implict covariant change to types of instance variables and method parameter and return types.

Thus BILINKABLE is not a subtype of LINKABLE.
Though BILINKABLE is internally consistent.