



Modeling with Inheritance

James Brucker



Uses of Inheritance

1. *Factor out common elements* (code reuse)

- parent class implements behavior needed by children
- parent defines *attributes* for all classes
- avoids duplicate or inconsistent code.

2. *Specialize*

- child class can **redefine** behavior of the parent

3. *Enable polymorphism*



Benefits of Inheritance?

1. Reuse code
2. Define a family of related types (polymorphism)



When To Use Inheritance?



Liskov Substitution Principle

In a program, if all objects of the superclass are replaced by objects from a subclass, the program should still work correctly.

Example:

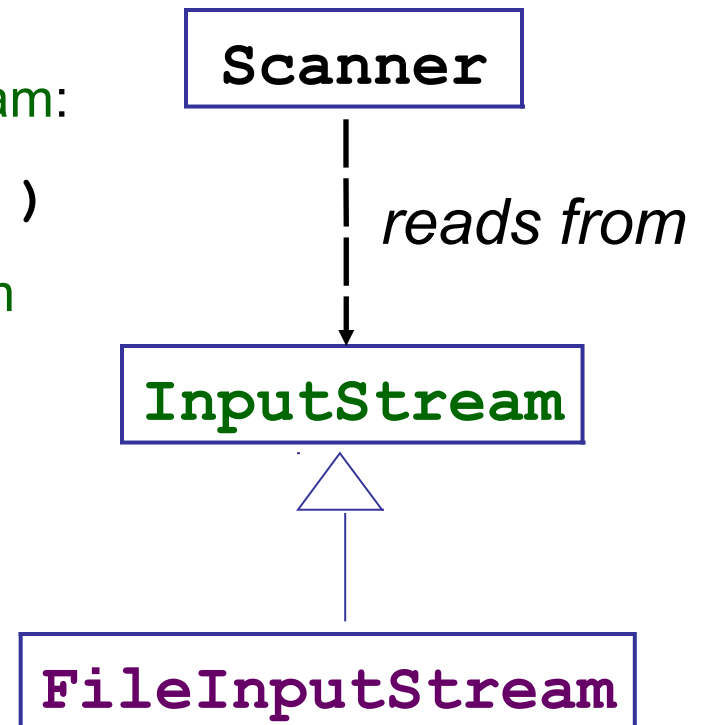
1. **Scanner** can read from an **InputStream**:

```
s = new Scanner ( InputStream )
```

2. **FileInputStream** extends **InputStream**

3. Scanner should also work correctly using a **FileInputStream**,

```
s = new Scanner( fileInputStream )
```





Liskov Principle example

A method that is expecting an object of a **ParentClass** type should also work if invoked with an object from any subclass.

```
public void doSomething( ParentClass obj )
```

should work with:

1. `doSomething (new ParentClass())`
2. `doSomething (new Subclass())`
3. `doSomething (new SubSubSubclass())`



Substitution Principle (2)

A Scanner can read from an `InputStream`

```
InputStream input = System.in;

// construct Scanner using InputStream
Scanner scanner = new Scanner( input );

while( scanner.hasNext() ) {
    String w = scanner.next( );
    System.out.print(w);
}
```



Substitution Principle (3)

Substitute a `FileInputStream` for the `InputStream`.

Scanner should still work!

```
InputStream input = null;
try {
    input = new FileInputStream(
        "/temp/sample.txt");
}
catch ( FileNotFoundException e ) { }
Scanner scanner = new Scanner( input );
while ( scanner.hasNext() ) {
    String w = scanner.next( );
    System.out.print(w);
}
```




Specialization

- A subclass can *override* (redefine) a method inherited from the parent, in order to *specialize* the behavior.
- Subclass *specializes* the behavior for its own needs, but still conforms to *contract* of parent's behavior.

```
public class Person {  
    private String name;  
    public String getName() { return name; }  
    public String toString() { return name; }  
}  
  
public class Student extends Person {  
    protected String studentId;  
    // redefine toString() to include Student ID.  
    public String toString() {  
        return getName()+" ID: "+studentId;  
    }  
}
```



Specialization *shadows* attributes

- If a child class defines an attribute with the same name as attribute of parent class, it creates a *new attribute* that "hides" the parent attribute.
- But the parent attribute is *still there* (in objects of child class)!
- This is called *shadowing*. Also called *hiding*.
- Use "**super**" to access parent's members.

```
public class Person {  
    protected long id;  
    . . .  
}  
  
public class Student extends Person {  
    private String id; // OK to define a new id!  
    public String toString() {  
        return "student id: " + id  
            + " citizen id: " + super.id;  
    }  
}
```



Shadow Attributes

- ❑ In general, **don't do it.**
- ❑ Redefining an attribute (deliberately) is poor design. Better to fix the design.
- ❑ duplicating an inaccessible, private variable (by coincidence) is OK.



Extension

- A subclass can define *new behavior* that the superclass does not have.
- A subclass can also define *new attributes*.

```
public class Person {  
    private String name;  
    public String toString() { return name; }  
}  
  
public class Student extends Person {  
    protected List<Course> courses; // new attribute  
  
    // new behavior  
    public void addCourse(Course c) { . . . }  
    public void getCredits( ) {  
        // compute credits for all courses that  
        // student got a passing grade.  
    }  
}
```



Bad Example: violates "is a" rule

A stack of objects is a simple data collection, like this...

<u>Stack</u>
+ push(Object) + pop() + peek() + isEmpty()

To store the data in the stack we could use a linked list...

<u>LinkedList</u>
+ addFirst(item) + addLast(item) + getFirst() + getLast() + get(index) + remove(index)

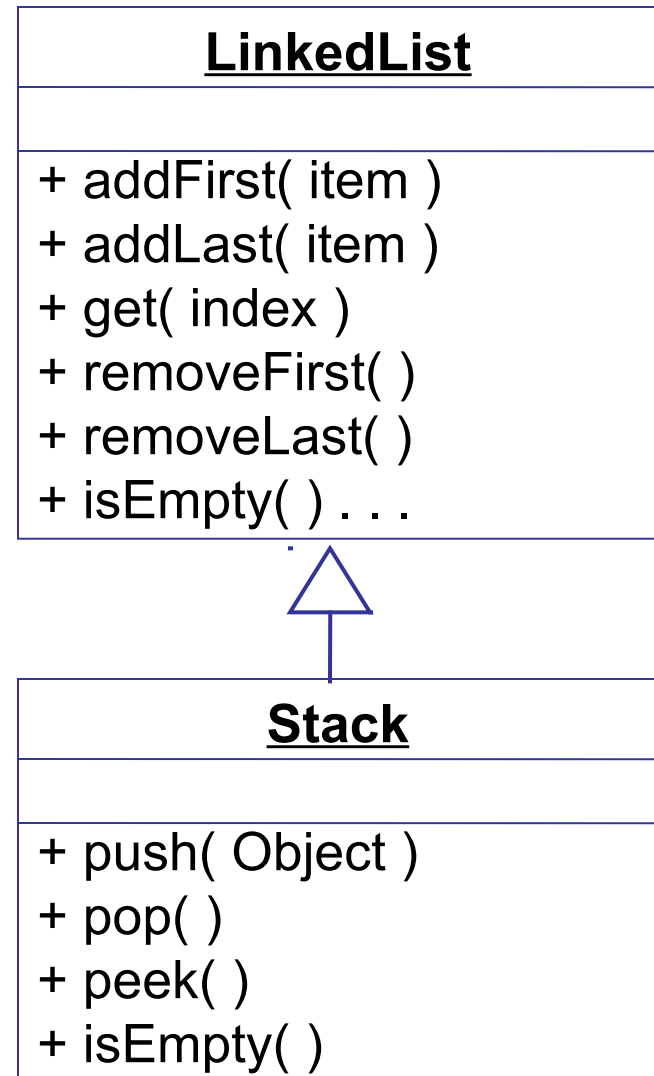


Example: A Stack (2)

Can we define Stack as a subclass of LinkedList?

All we need to do is add the 4 stack methods and we're done!

```
class Stack
    extends LinkedList {
    public Stack() { super(); }
    public void push(Object o) {
        addFirst( o );
    }
    public Object pop() {
        return removeFirst( );
    }
    public Object peek() {
        return get(0);
    }
}
```





Example: A Stack (3)

The problem with this is that Stack will exhibit **all** the behavior of a LinkedList, including methods that should not exist for a stack.

```
/* Stack example */
public void stackTest( ) {
    Stack stack = new Stack( );
    stack.push( "First item" );
    stack.push( "Second item" );
    stack.push( "Third item" );
    stack.push( "Fourth item" );
    // cheat! get the 3rd item
    String s = stack.get( 2 );
    // cheat! add item at front of stack
    stack.addFirst( "Ha ha ha!" );
```

Behavior inherited
from LinkedList



"is a" (kind of) relationship

A simple test for whether inheritance is reasonable:

*Subclass **is a** Superclass*

- ❑ CheckingAccount **is a** (kind of) BankAccount
- ❑ Number **is an** (kind of) Object
- ❑ Double **is a** (kind of) Number
- ❑ Rectangle is a 2-D Shape
 - ✓ **Rectangle extends Shape2D**



"is a" test **doesn't** always work

X A Square **is a** Rectangle

but a rectangle can have length \neq width

X ArrayList **is a** List

List is a *type* (interface) not a class

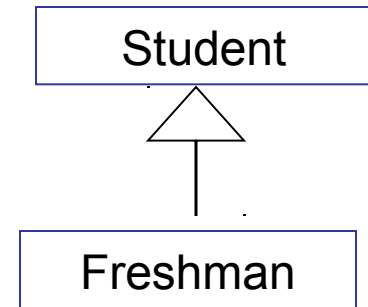
X A Freshman **is a** Student

but next year she will be a sophomore.

✓ Use an attribute for features that **change**.

X George Bush **is a** President

an *instance* of a class, not a subclass



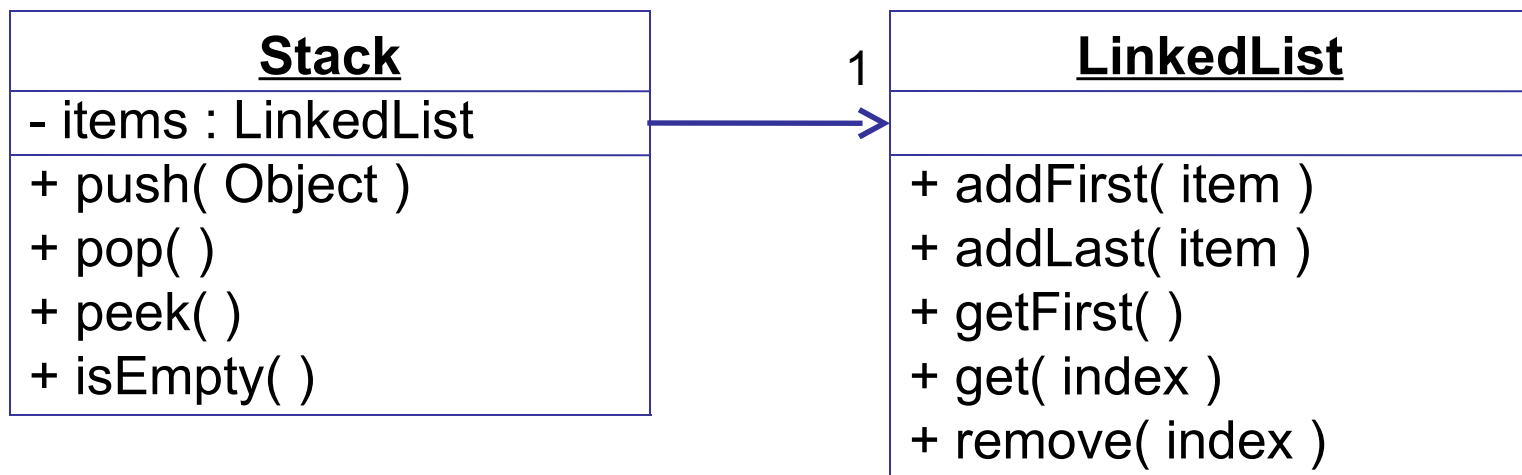


Attribute: "has a"

- In the case of a Stack, we would say:

*"a Stack **has a** LinkedList"*

- "**has a**" means that something should be an attribute
- "has a" indicates an association.
- UML uses an open arrowhead for association

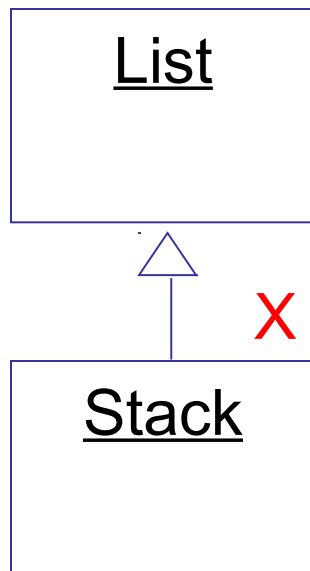




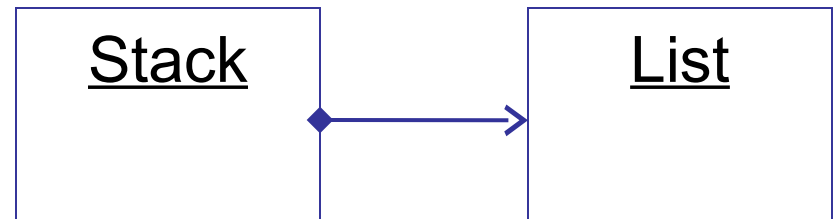
Composition vs Inheritance

"Favor composition over inheritance"
(design principle)

Consider using **aggregation** (has a ...) instead of **inheritance** (is a ...).



X *Stack is a List*



✓ *Stack has a List*



Java LinkedList is a Stack !

- ❑ java.util.LinkedList provides the stack methods.
- ❑ You can use LinkedList as a Stack or Deque.
- ❑ **Is this a good design?**

Stack methods are
defined in LinkedList



<u>LinkedList<T></u>
+ addFirst(item) + addLast(item) + getFirst(): T + get(index): T + remove(index) + isEmpty() + push(item) + pop(): T + peek(): T



Problems with Inheritance

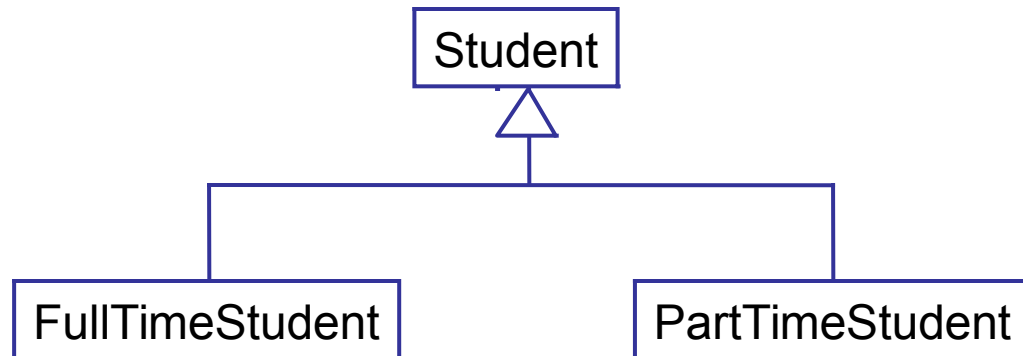
- Can only have **one** parent class
 - **Binds** objects to one hierarchy (not flexible)
 - Sometimes the parent class *doesn't know how* a behavior should be implemented
- Example:** Shape is parent for Rectangle, Circle, ...
what should `Shape.draw()` do?

Don't Overuse Inheritance...

- Don't use a subclass in situations where an object may need to change class during its life time.

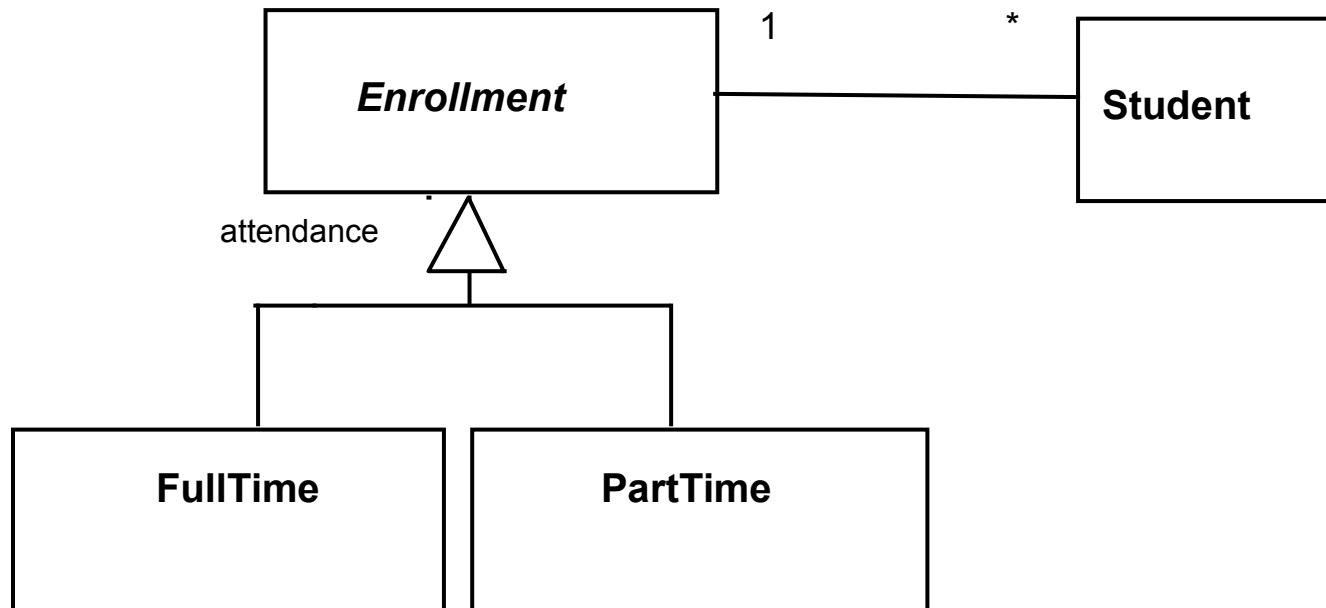
Example: Full-time and Part-time students have different requirements and behavior.

Should we model this using inheritance?



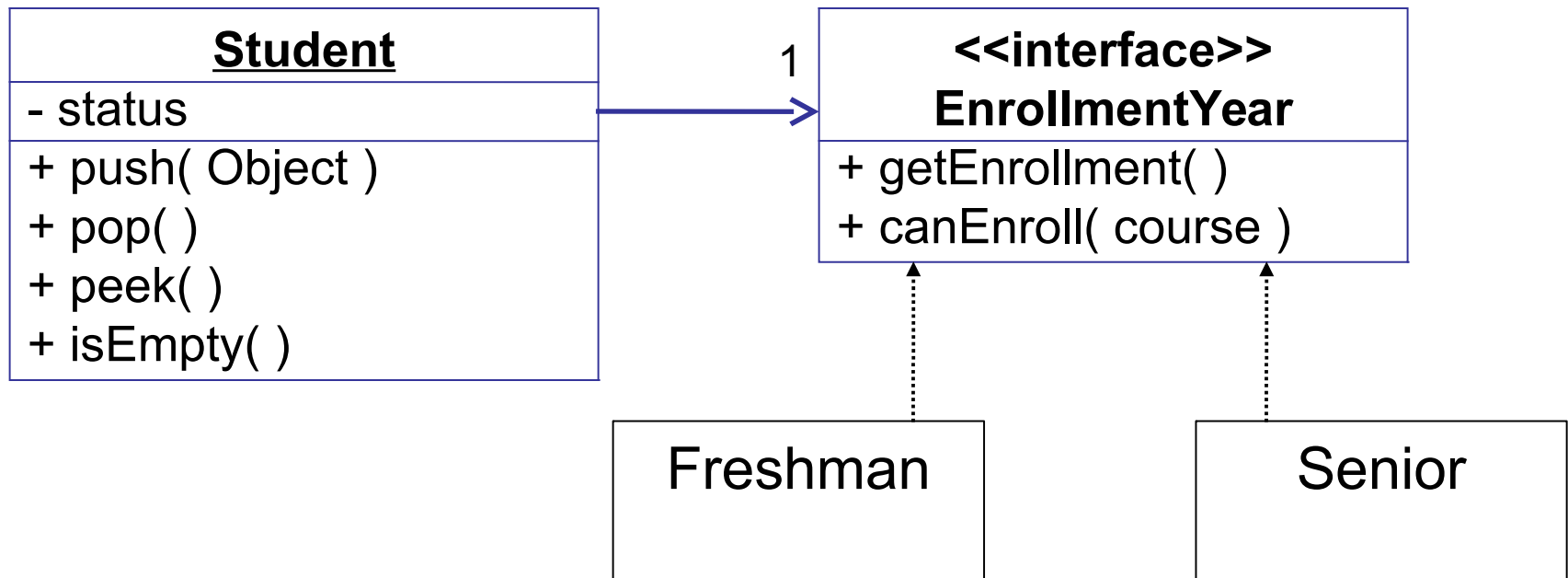
Modeling a "role" or "status"

- A student can **change** from **Full-time** to **Part-time**.
- Full-time or part-time is a *role* or *status*.
- Model this using an **attribute** that refers to an object of the appropriate status.



Prefer attribute over inheritance

- For Freshman - Student example, model Freshman as an attribute... and assign behavior to it.

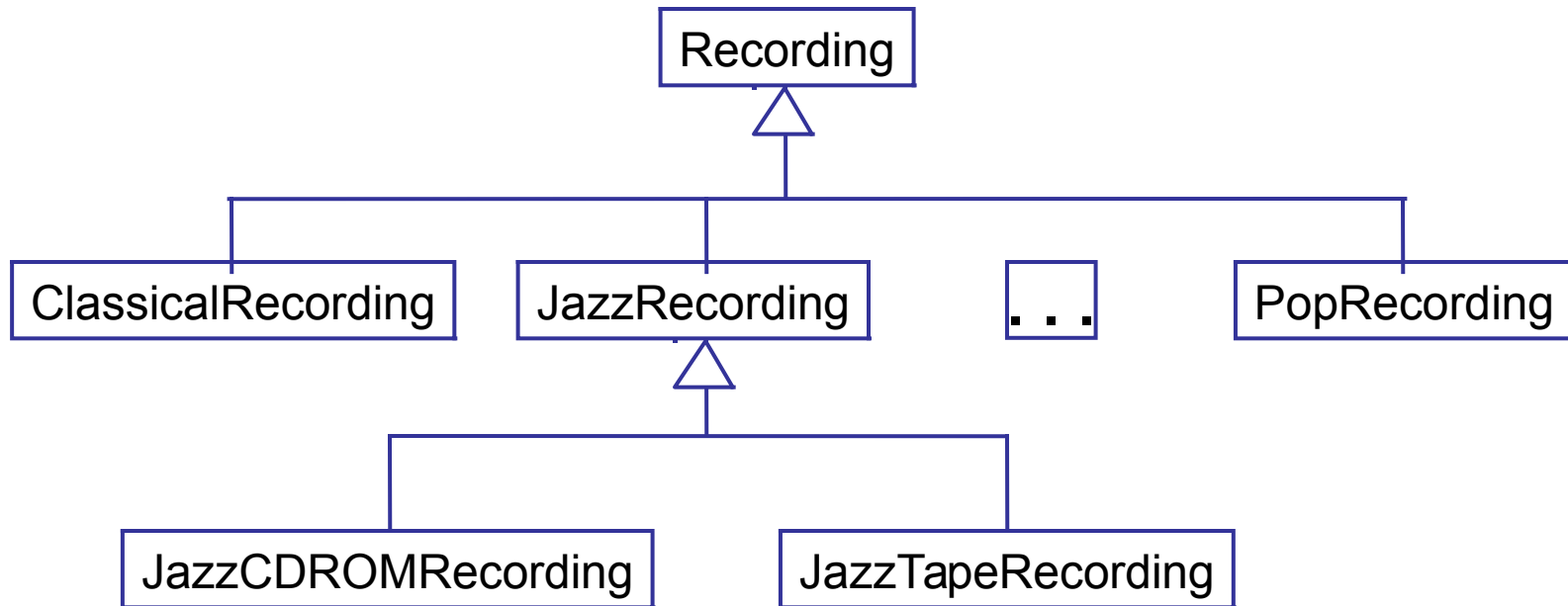


Don't Overuse Inheritance...

- Subclass doesn't add significant extension or specialization

Example: a library has several types of recordings: Jazz Recording, Classical Recording, Pop Recording, ...

Recordings may be Tape or CDROM



Use Interface to describe behavior

- "Set" is an **interface** because it doesn't *implement* any methods or provide any *attributes*.
- HashSet *implements* Set

