Modeling with Inheritance

James Brucker

Uses of Inheritance

Factor out common elements (code reuse)

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

Specialize

- child class can redefine behavior of the parent

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:

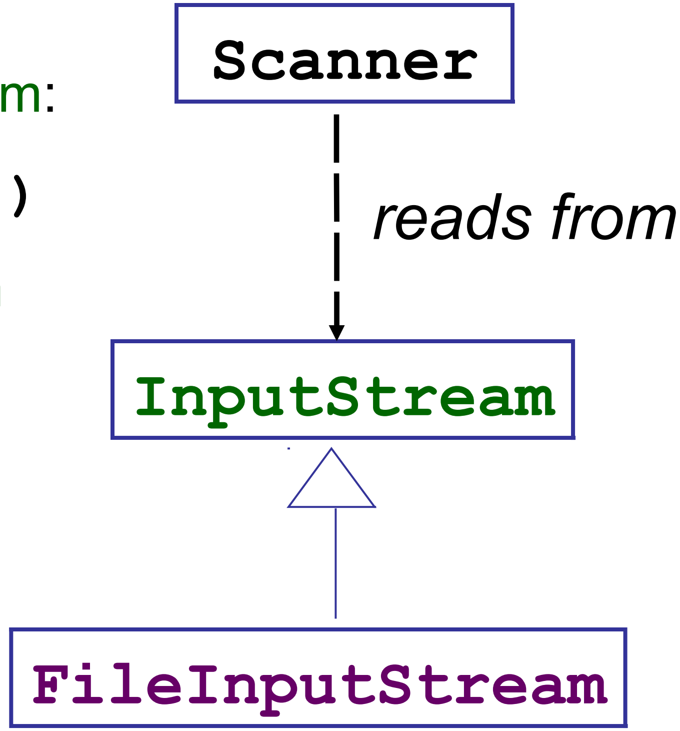
1. Scanner can read from an InputStream:

   s = **new Scanner(InputStream)**

2. FileInputStream extends InputStream

3. Scanner should also work correctly using a FileInputStream,

   s = new Scanner(fileInputStream)



Substitution Principle

Any code that is expecting an object of a Superclass 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)

Construct a Scanner using an `InputStream`

```
Scanner scanner;
InputStream instream = System.in;

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

while( scanner.hasNext() ) {
   String w = scanner.next( );
   . . .
```

Substitution Principle (3)

Substitute a FileInputStream for the InputStream.

Scanner should still work!

```
String FILENAME = "/temp/sample.txt";
Scanner scanner = null;
try {
  InputStream instream =
      new FileInputStream( FILENAME );
  scanner = new Scanner( instream );
} catch ( FileNotFoundException e ) {  }

while (scanner.hasNext()) {
   String w = scanner.next( );
   . . .
```

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 {
        protected String name;
        public String toString() { return name; }
}
public class Student extends Person {
        protected String studentID;
        // redefine toString() to return our ID, too.
        public String toString() {
                return name+" "+studentID;
        }
}
```

Specialization & Access Permissions

- A subclass cannot "reduce visibility" of a **method** it redefines from parent:

```
public class Person {
       private String toString() { // ERROR
         return "I like some privacy";
    }
```

Visibility in Parent Class	Visibility in Child Class
`public String toString( )`	`public`
`protected void setName( )`	`protected` or `public`
`private int getRadix( )`	*anything* -- private method is statically bound, visible only inside parent class.
(default) `getName( )`	Homework

Substitution explains the visibility rule

a consequence of the *Substitution Principle*

"*An object of a child class can be substituted any where that an object of the parent class is expected*"

```
public class Printer {
       public void display(Object obj) {
              System.out.println( obj.toString() );
       }
}
```

This works with Object, so it must work correctly with any sublcass of Object

Substitution explains visibility rule

```
public class Person {
    protected String name;
    protected long ID;
    public long getID() { return ID; }
}

public class Student extends Person {
    private String ID;  // OK to redefine ID!

// Illegal! (1) less visible, (2) change type
    protected String getID() { return ID; }
```

Specialization *shadows* attributes

- Parent's data members and methods are *not replaced* by child's members, they are simply *shadowed.*
- Use "**super**" to access parent's members.

```
public class Person {
    protected String name;
    protected long ID;
    public long getID() { return ID; }
}

public class Student extends Person {
    private String ID;  // OK to redefine ID!

    public Student() {
        super.ID // refers to parent ID
```

Shadow Attributes

- In general, **don't do it**.
- If you *need* to shadow an attribute (to change it), its a sign of poor design. Better to fix the design.

Extension

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

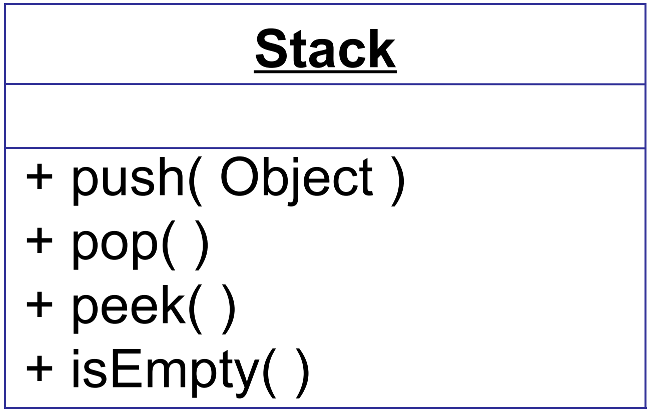
```
public class Person {
   protected String name;
   public String toString() { return name; }
}

public class Student extends Person {
   protected int credits;      // new attribute

   // new behavior
   public void addToCredits(int n) { credits += n; }
   public void getCredits( ) { return credits; }
   ...etc...
}
```

Example: A Stack

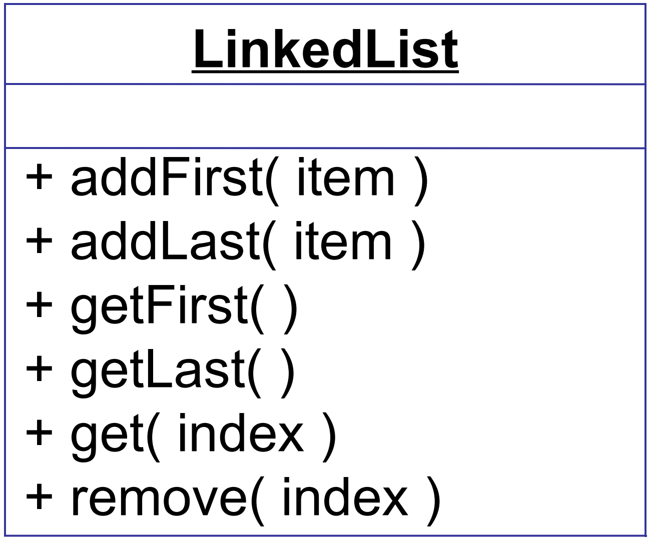
A stack of objects is a simple data collection, like this...



Stack

+ push(Object)
+ pop( )
+ peek( )
+ isEmpty( )

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



LinkedList

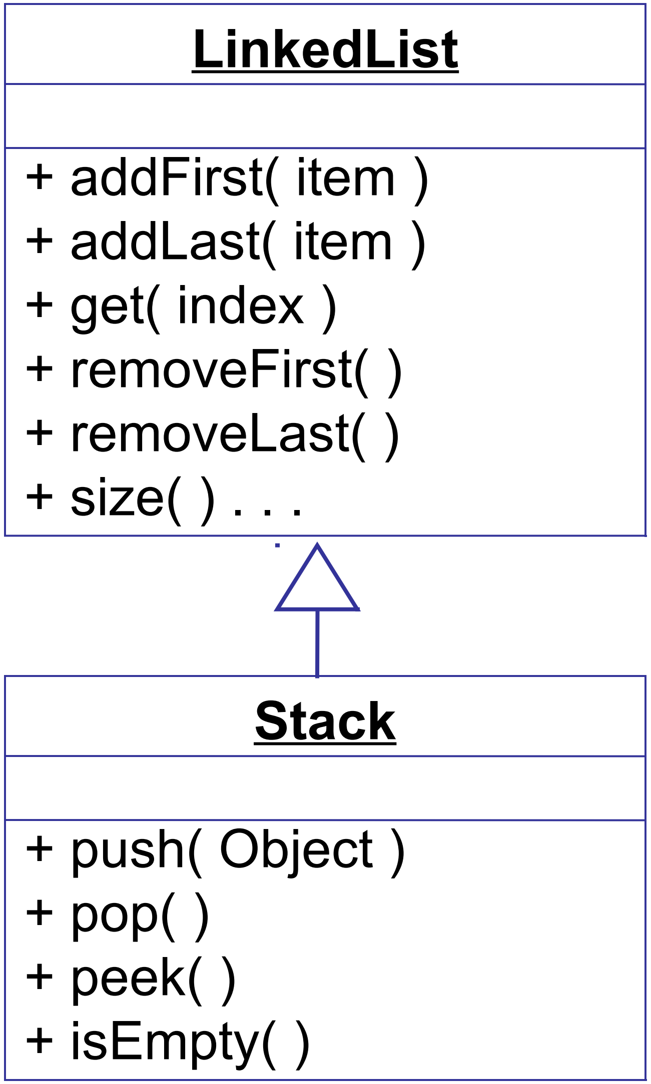
+ addFirst(item)
+ addLast(item)
+ getFirst( )
+ getLast( )
+ get(index)
+ remove(index)

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){
    addLast( o );
  }
  public Object pop() {
`   return removeLast( );
   }
  public boolean isEmpty(){
    return super.size()==0;
  }
```



LinkedList

+ addFirst(item)
+ addLast(item)
+ get(index)
+ removeFirst( )
+ removeLast( )
+ size( ) . . .

Stack

+ push(Object)
+ pop( )
+ peek( )
+ isEmpty( )

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 3nd item
       String s = stack.get( 2 );
       // cheat! add item at front of
stack
       stack.addFirst( "Ha ha ha!");
```

Behavior inherited
from ListedList

"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 ≠ width

X ArrayList is a List

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

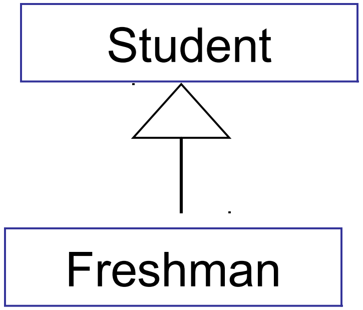
X A Freshman is a Student

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



Student

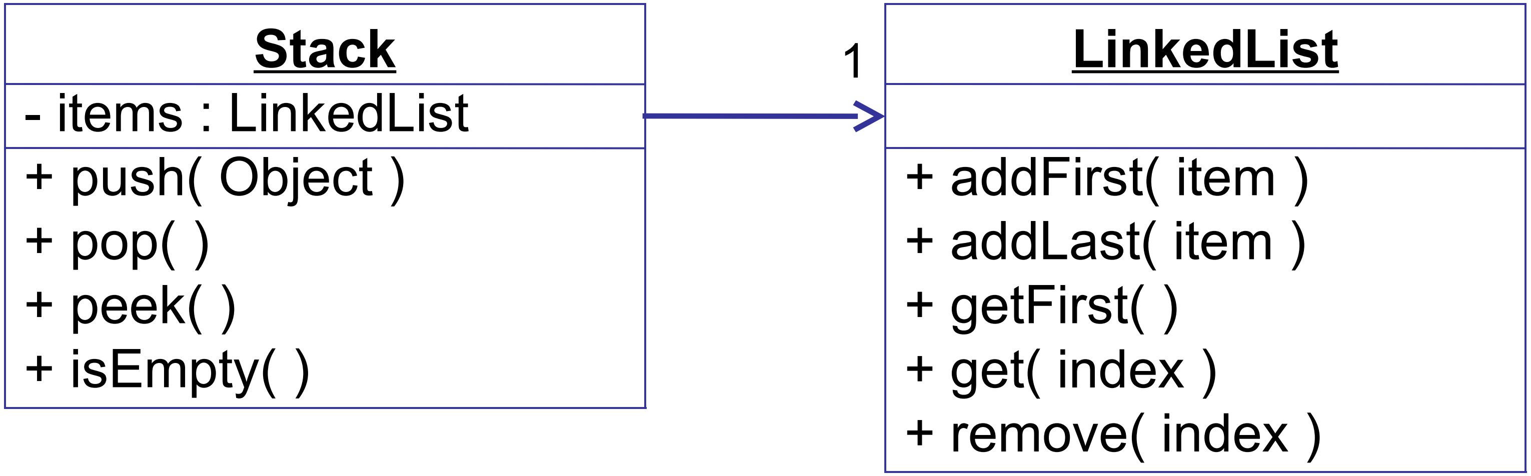
Freshman

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



Stack

- items : LinkedList

+ push(Object)
+ pop( )
+ peek( )
+ isEmpty( )

1

LinkedList

+ addFirst(item)
+ addLast(item)
+ getFirst( )
+ get(index)
+ remove(index)

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`, ...

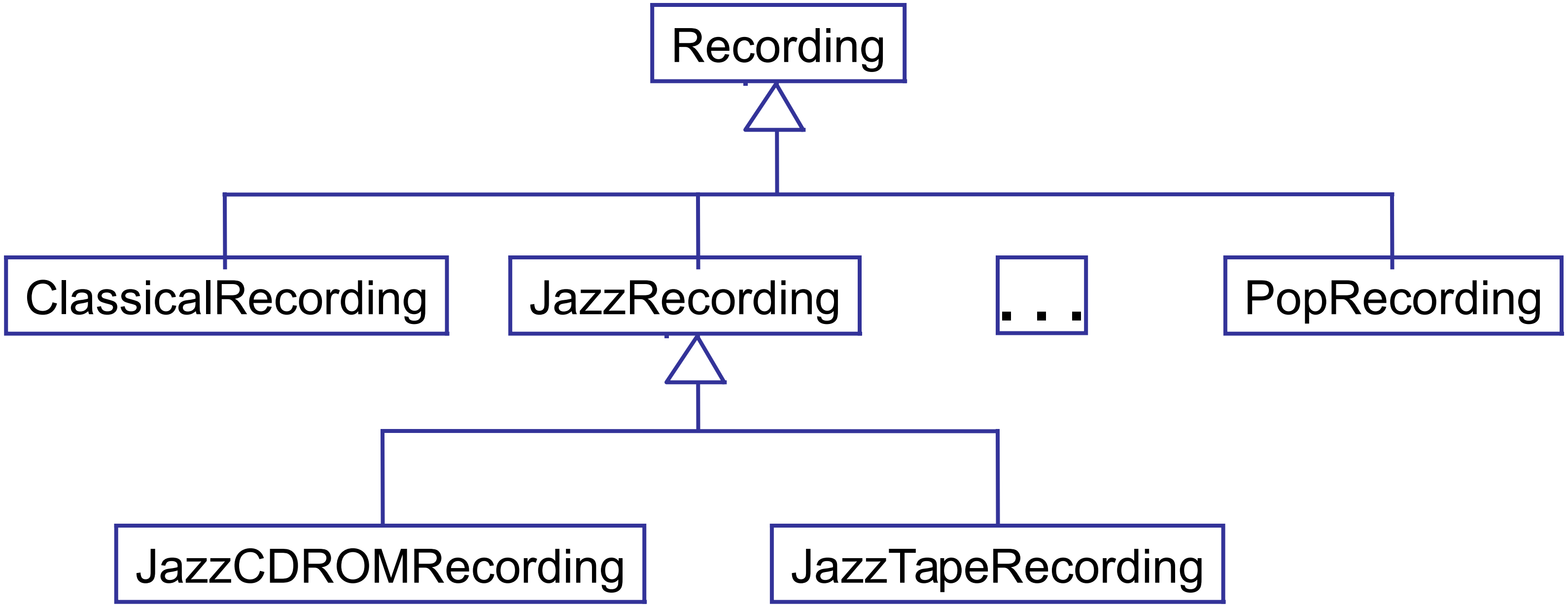
  what should `Shape.draw( )` do?

Don't Overuse Inheritance...

- 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

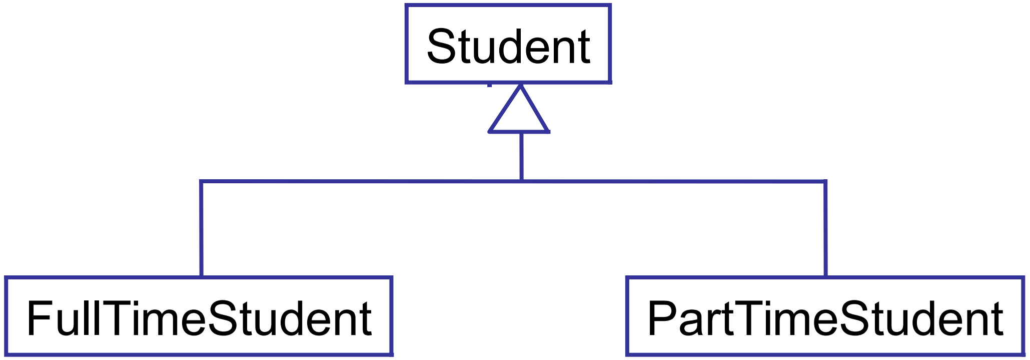


Don't Overuse Inheritance...

- 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?

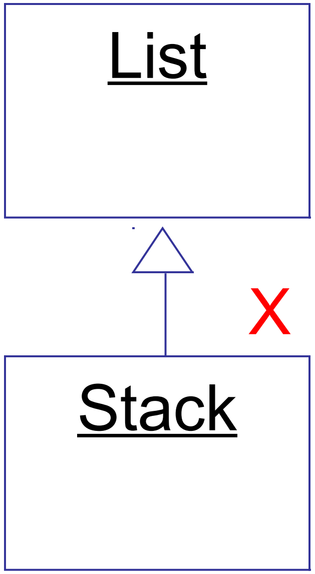


Composition vs Inheritance

"*Favor composition over inheritance*"

(*design principle*)

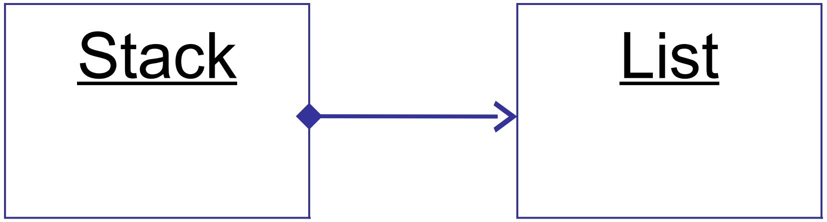
Consider using aggregation (has a ...) instead of inheritance (is a ...).



List

Stack

X *Stack is a List*



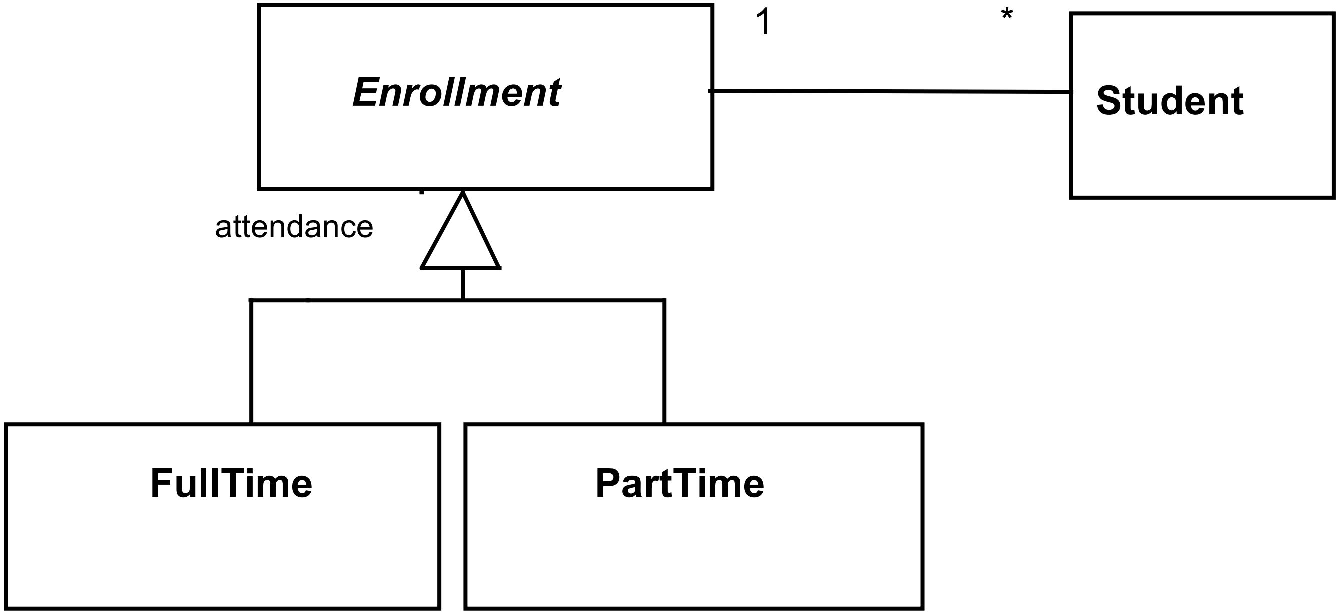
Stack

List

✓ *Stack has a List*

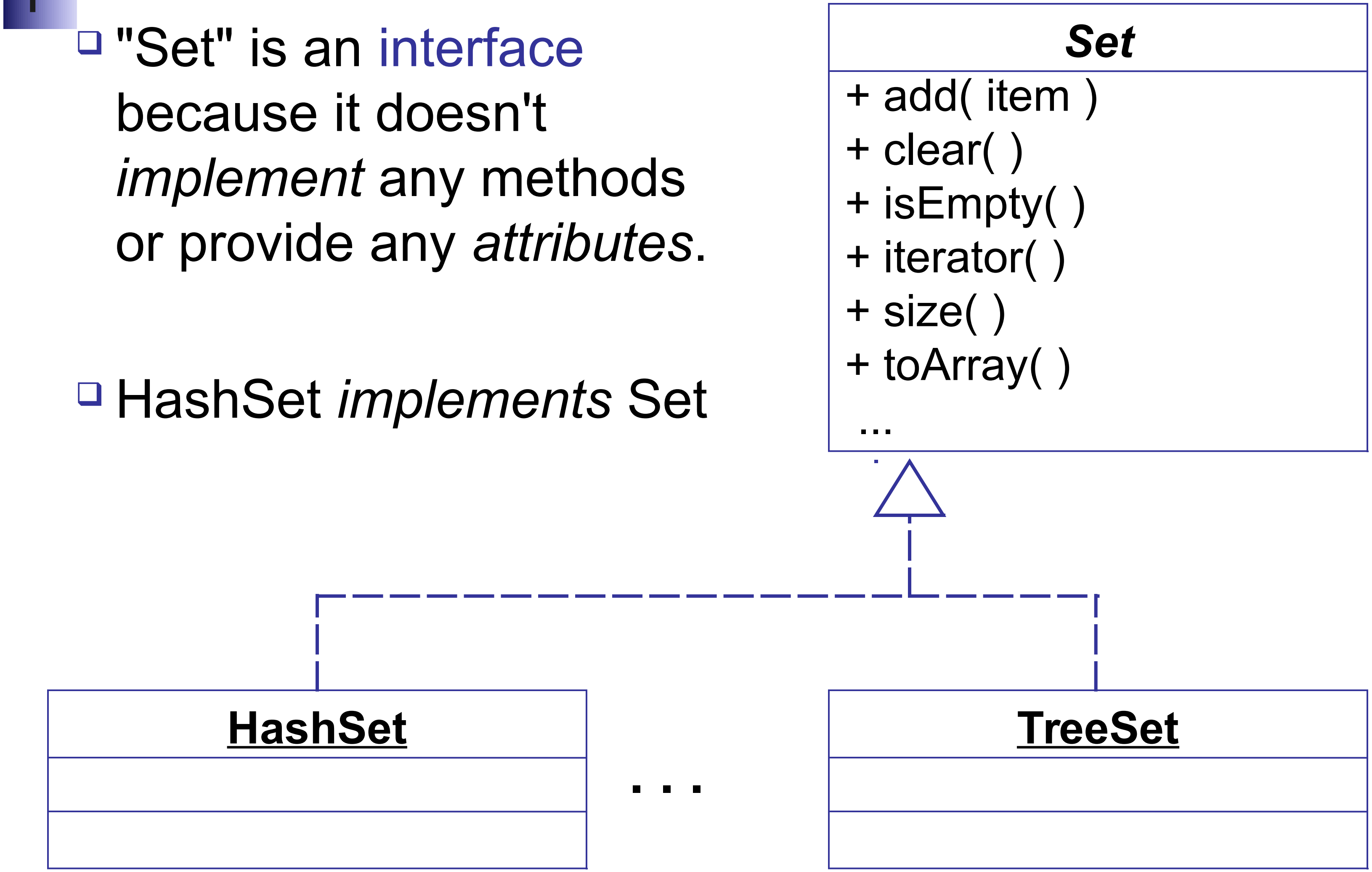
Modeling a "role" or "status"

- 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.



Use Interface to describe behavior

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



Set

+ add(item)
+ clear( )
+ isEmpty( )
+ iterator( )
+ size( )
+ toArray( )
...

HashSet

. . .

TreeSet