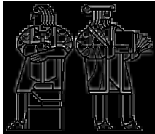


6.170 Lecture 10 Subtyping & Subclassing



John Chapin
MIT EECS
2-22-2000

6.170

Outline

1. Subtype polymorphism
2. Theory of subtypes
3. Subclassing

© John Chapin/John Guttag

2

6.170

What is subtyping?

Sometimes every A is a B

library database: every book and CD is a library holding

Subtyping expresses this in the program

programmer declares A is a subtype of B

meaning: "every object that satisfies interface A
also satisfies interface B"

**Goal: code written using B's specification operates
correctly even if given an A**

© John Chapin/John Guttag

3

6.170

What is an interface?

interface:

a group of method specifications

a contract at the boundary between modules

using program ↔ ADT implementation

application ↔ operating system

ATM ↔ bank computer

satisfy an interface

provide an implementation that satisfies each of the
specifications

© John Chapin/John Guttag

4

6.170

Java terminology

```
interface I extends J { }  
I and J contain signatures only, no implementations  
I is a subtype of J
```

```
class A implements I { }  
objects of class A satisfy interface I  
A is an implementation of I
```

```
class A extends B { }  
objects of class A reuse the implementation of B  
A is a subclass of B  
type of A is also a subtype of type of B
```

```
abstract class C { }  
C has unimplemented signatures
```

© John Chapin/John Guttag

5

6.170

Subtyping example

```
interface GObject {           // this is the SUPER type  
    void draw();  
    // effects: paint this on the screen  
    Rectangle boundingBox();  
    // effects: return the region within which  
    //           this is painted when draw() is called.  
}  
  
interface ColorObj extends GObject { // SUB type  
    void setColor(Color c); // effects: modify this's color  
    Color getColor();       // effects: return this's color  
}  
  
interface TextField extends GObject {  
    String getText();  
    // effects: return the text currently stored in this  
}
```

© John Chapin/John Guttag

6

Java details of subtyping

ColorObj includes all GObj methods

```
interface ColorObj extends GObj {
    void setColor(Color c);
    Color getColor();
    void Draw();
    Rectangle boundingBox();
    // java adds these lines if you don't write them
}
```

All ColorObjs may be assigned to GObj variables

```
ColorObj co = new ColorObj1();
GObj g = co;    // Always OK
```

If a GObj variable refers to a ColorObj, can cast it

```
ColorObj co2 = (ColorObj) g;
// throws ClassCastException if not a ColorObj
```

Polymorphism: avoids type case

```
Vector objsOnScreen;
void drawAll() {
    Iterator i = objsOnScreen.iterator();
    while (i.hasNext()) {
        GObj g = (GObj) i.next();
        g.draw();
    }

    void drawAll() {
        Iterator i = objsOnScreen.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            if (o instanceof ColorObj) {
                ColorObj c1 = (ColorObj) o;
                c1.draw();
            } else if (o instanceof TextField) {
                TextField t1 = (TextField) o;
                t1.draw();
            }
        }
    }
}
```

Subtype polymorphism benefits

Benefits to code that uses GObj:

code reuse

code applies without change to ColorObj, TextField
new subtypes can be added without change

reduced complexity

programmer need only understand GObj interface,
not ColorObj, TextField details

better modularity

no dependency if ColorObj or TextField changes

Part 2: Theory of subtyping

Does A extends B always imply A is a subtype of B?

Example:

```
interface ColorObj extends GObj {
    ...
    void Draw();
    // requires: the color of this has been set
    // effects: paint this on the screen
}

void drawAll() {
    Iterator i = objsOnScreen.iterator();
    while (i.hasNext()) {
        GObj g = (GObj) i.next();
        g.draw();
    }
}
```

Effects of restricting ColorObj.draw()

Response 1: no change to application

Application doesn't set color in some ColorObj
drawAll() fails
BAD

Response 2: set color before storing in objsOnScreen

rationale for this design not visible in code
can forget the hidden dependency when modifying
BAD

Problem:

GObj.draw() specification not sufficient to use
ColorObj.draw() safely
ColorObj is not a GObj

Your turn

Elementary school: every square is a rectangle

```
interface RectangleObj extends GObj {
    void setSize(int w, int h) { ... }
    // effects: this_post.width = w,
    //           this_post.height = h
}

interface SquareObj extends RectangleObj { ... }
```

Choose the best option for SquareObj.setSize

- void setSize(int w, int h);
// requires: w = h
- void setSize(int edgeLength);
- void setSize(int w, int h) throws BadSizeException
// throws BadSizeException if w != h



End of part 2

Subtype methods must be substitutable for supertype methods

- No more exceptions
- No more requires
- Modifies less

This occasionally violates our intuitions



Part 3. Subclassing

Most claimed benefits of subclassing are actually from subtyping

- code reuse (client)
- reduced complexity
- better modularity

Additional feature: code reuse (implementation)
usually increases complexity