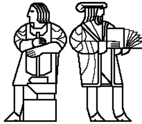
6.170 Lecture 10
Subtyping and subclassing



Michael Ernst
MIT EECS

Outline

Subtype polymorphism
Theory of subtypes
Subclassing
- What is subclassing
- Subclassing vs. subtyping
- Potential dangers of subclassing

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

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

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

Subtyping example

```
interface GObj {            // this is the SUPER type
   void draw();
   // effects:  paint this on the screen
   Rectangle boundingBox();
   // returns:  the region within which
   //           this is painted when draw() is called.
 }

interface ColorObj extends GObj {   // SUB type
   void setColor(Color c);// effects: this.color = c
   Color getColor();      // returns: this.color
 }

interface TextField extends GObj {
   String getText();
   // returns: the text currently stored in this
 }
```

Java details of subtyping

ColorObj includes all GObj methods

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

All ColorObjs may be assigned to GObj variables

```
ColorObj co = new ColorObj1();  // concrete subtype
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

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

```
1. void setSize(int w, int h);
      // requires: w = h
2. void setSize(int edgeLength);
3. void setSize(int w, int h) throws BadSizeException
      // throws BadSizeException if w != h
```

Subtyping summary

Subtype methods must be substitutable for supertype methods
- No more exceptions
- No more requires
- No more modifies

This occasionally violates our intuitions

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

What you need to know

What you need to know
- Difference between subtyping and subclassing
- Difference between true subtyping and Java subtyping
- Potential dangers of subclassing
- Purpose of a Java interface

The answers

A is a subclass of B
- Every A object has a B object inside of it

A is a Java subtype of B
- a variable of type B may refer to an object of type A

A is a true subtype of B
- all code that operates correctly on objects of type B will operate correctly on objects of type A

A is a subclass of B → A is a Java subtype of B

NOT (A is a Java subtype of B → A is a true subtype of B)

What is subclassing?

```
                                  // the SUPER class
class Bbox1 {
  Rectangle bbox;
  void updateBBox(int w, int h)        { ... }
  public Rectangle boundingBox()       { return bbox; }
}
                                  // the SUB class
class Circle1 extends Bbox1 {
  Point center;
  int radius;
  public void draw() throws Offscreen { ... }
}
```


Basic idea of subclassing

Every Circle1 object incorporates a Bbox1 object

```
Bbox1 b1 = new Bbox1();     Circle1 c1 = new Circle1();
```

Inheritance

Use a Bbox1 method if Circle1 doesn't provide it

c1
center
radius
draw
bbox
updateBBox
boundingBox

```
Circle1 c1 = new Circle1();
c1.updateBBox(12,6); // OK
```

Circle1 inherits the method and fields from Bbox1

Java: subclassing implies subtyping

Consider:

Every Circle1 object has a Bbox1 "inside" of it
All Bbox1 methods inherited by Circle1
∴ Each Circle1 object supports all Bbox1 functionality
∴ Circle1 may be a subtype of Bbox1

Java supports this

```
Circle1 c1 = new Circle1();
Bbox1 b = c1;        // OK

  // compare to:
String s1 = "hello";
Bbox1 b2 = s1;       // compile-time error
```


Overriding

Use subclass method if both classes provide it

c2 b
updateBBox A
center
radius
draw
bbox
updateBBox B
boundingBox

```
Circle2 c2 = new Circle2();
c2.updateBBox(12,6); // A runs

Bbox1 b = c2;
b.updateBBox(12,6);  // A runs
```

Circle2 overrides the method B inherited from Bbox1.

Potential benefits of subclassing

Implementation reuse

Implement Bbox1.updateBBox() once
Reuse for Circle1, Square1, etc.

Modularity

Can ignore private fields and methods of superclass when working on subclass

All subtyping benefits

The answers

A is a subclass of B
Every A object has a B object inside of it

A is a Java subtype of B
a variable of type B may refer to an object of type A

A is a true subtype of B
all code that operates correctly on objects of type B will operate correctly on objects of type A

A subclass of B → A Java subtype of B

NOT (A Java subtype of B → A true subtype of B)

Model of types as sets

Object
GObj
ColorObj

ColorObj subset of GObj:
$o \in$ ColorObj $\rightarrow o \in$ Gobj
Code that operates correctly on GObj objects will operate correctly on ColorObj objects

Java subtyping

```
interface GObj { ... }
interface ColorObj extends GObj { ... }
class Bbox1 implements GObj { ... }
class Triangle1 implements ColorObj { ... }
```

Object
GObj
Bbox1
ColorObj
Triangle1

Gobj is abstract : all objects ∈ Gobj are ∈ some subset

More powerful subtyping

```
interface GObj { ... }
interface ColorObj extends GObj { ... }
interface Textfield extends GObj { ... }
class CText implements Textfield, ColorObj {}
```

Object
GObj
ColorObj
Textfield
CText

CText a subset of both Textfield and ColorObj

Java interfaces

A Java interface enables the programmer to

declare a set

```
interface GObj {
```

describe the properties of members of that set

```
        Rectangle boundingBox();
        void draw();
}
```

declare which objects belong to that set

```
class Circle1 implements GObj { ... }
```

declare objects which belong to multiple, unrelated sets

```
class Movie implements Gobj,Animation {}
```


Why interfaces instead of classes

Java design decisions:

A class has exactly one superclass
A class may implement multiple interfaces
An interface may extend multiple interfaces

Observation:

multiple superclasses are difficult to use and to implement
multiple interfaces, single superclass gets most of the benefit

Problems with subclassing

If not careful...

Subclass ends up dependent on superclass details
Superclass ends up dependent on subclass details

Result

Superclass and subclass behave as one big module
Bad engineering

Subclass depends on superclass

```
class Circle1 extends Bbox1 {
  public void draw() throws Offscreen {
   if (bbox.xmin < 0) { ... }
   ...
  }
}
```

Circle1 can access the implementation of Bbox1

complex interface between Circle1 and Bbox1
Circle1 usually must change when Bbox1 changes

Superclass depends on subclass

```
class Bbox1 {
   void updateBBox()   {
       int xmin = computeXmin();
       ...
   }
   int computeXmin() { return -1; }
}

class Circle1 extends Bbox1 {
   ...
   int computeXmin() { ... }
}
```

Unexpected overriding
- Bbox1 calls itself, gets Circle1 method instead
- ∴ must study Circle1 to understand Bbox1 fully

Avoiding these problems

Very careful design required
- Make subclass interface explicit
- Keep it simple
- Mark everything else private
- Test overriding methods thoroughly
- Use safe overriding where feasible

Safe use of overriding

```
class Circle2 extends Bbox1 {
   Point center;
   int radius;
   public void draw() throws Offscreen { ... }
   public void updateBBox(int w, int h) {
       super.updateBBox(w, h);
       radius = min(w/2, h/2);
   }
}

   Bbox1 b = new Circle2();
   b.updateBBox(6, 12);
```

Bbox1 method executes as part of Circle2 method