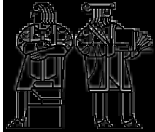


6.170 Lecture 11 Subclassing and Subtyping part 2



John Chapin
MIT EECS
2-23-2000

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

Outline

1. What is subclassing
2. Subclassing vs. subtyping
3. Potential dangers of subclassing

Original content of today's lecture:

- read sections 7.4 - 7.6
- skim section 7.9 (understand basic idea, don't need to memorize details)

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)

Part 1. What is subclassing

```

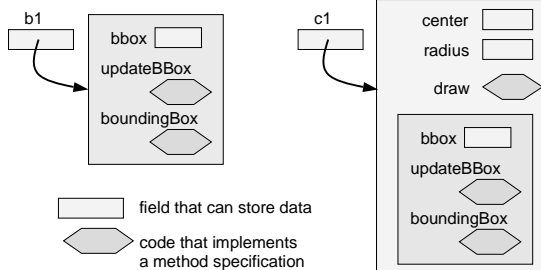
// the SUPER class
class Bbox1 {
    Rectangle bbox;
    void updateBBox(int w, int h) { ... }
    public Rectangle boundingBox() { return bbox; }
}

// the SUB class
class Circle1 extends Bbox1 {
    Point center;
    int radius;
    public void draw() throws Offscreen { ... }
}
    
```

Basic idea of subclassing

Every Circle1 object has a Bbox1 object inside of it

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

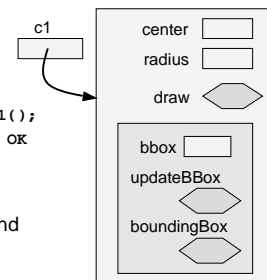


Inheritance

Use a Bbox1 method if Circle1 doesn't provide it

```
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 is probably 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
```

© John Chapin/John Guttag 7

Overriding

Use subclass method if both classes provide it

```
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.

© John Chapin/John Guttag 8

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

© John Chapin/John Guttag 9

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)

© John Chapin/John Guttag 10

Part 2. Model of types as sets

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

© John Chapin/John Guttag 11

Java subtyping

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

GObj is abstract : all objects \in GObj are \in some subset

© John Chapin/John Guttag 12

More powerful subtyping

```

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

```

Object

CText a subset of both Textfield and ColorObj

© John Chapin/John Guttag 13

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 {}

```

© John Chapin/John Guttag 14

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

© John Chapin/John Guttag 15

Part 3. 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

© John Chapin/John Guttag 16

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

© John Chapin/John Guttag 17

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

© John Chapin/John Guttag 18

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