CSE 331
Software Design & Implementation

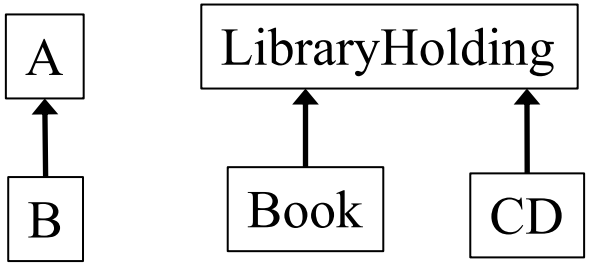
James Wilcox

Autumn 2021

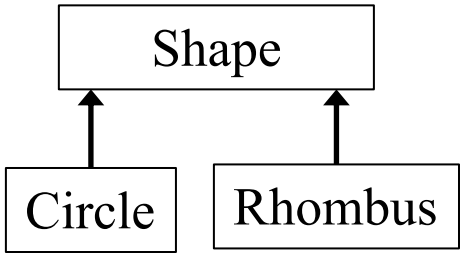
Subtypes and Subclasses

What is subtyping?

Sometimes "*every B is an A*"
- examples in a library database:
  - every book is a library holding
  - every CD is a library holding



For subtyping, "*B is a subtype of A*" means:
- "every object that satisfies the rules for a B also satisfies the rules for an A"
- (B is a strengthening of A)



Goal: code written using A's **spec** operates correctly if given a B
- plus: clarify design, share tests, (sometimes) share code

Subtypes are substitutable

Subtypes are ***substitutable*** for supertypes

- Liskov substitution principle
- instances of subtype won't surprise client by **failing to satisfy** the supertype's specification
- instances of subtype won't surprise client with **more expectations** than the supertype's specification

We say B is a ***(true) subtype*** of A if B has a stronger specification than A

- (or is equally strong)
- this is ***not*** the same as a ***Java* subtype (e.g. subclass)**
- Java subclasses that are not true subtypes: *confusing* & *dangerous*
  - but unfortunately common ☹
  - Java allows casting sub- to supertypes assuming true subtypes

Subtyping vs. subclassing

Substitution (subtype) is a matter of specifications

- B is a subtype of A iff an object of B can masquerade as an object of A in any context
- B is a subtype if its spec is is a strengthening of A's spec

Inheritance (subclass) is a matter of implementations

- factor out repeated code
- to create a new class, write only the differences

Java purposely merges these notions for classes:

- every subclass is a Java subtype
- but not necessarily a true subtype
- (though Java casting rules **assume** true subtypes)

Inheritance makes adding functionality easy

Suppose we run a web store with a class for *products…*

```
class Product {
    private String title;
    private String description;
    private int price; // in cents
    public int getPrice() {
        return price;
     }
    public int getTax() {
        return (int)(getPrice() * 0.086);
    }
    …
}
```

... and we need a class for *products that are on sale*

Copy and Paste

```
class SaleProduct {
    private String title;
    private String description;
    private int price; // in cents
    private float factor;
    public int getPrice() {
       return (int)(price*factor);
    }
    public int getTax() {
        return (int)(getPrice() * 0.086);
    }
    …
}
```

Not a good choice. — Why? (hint: properties of high quality code)

Inheritance makes small extensions small

Better:

```
class SaleProduct extends Product {
    private float factor;
    public int getPrice() {
      return (int)(super.getPrice()*factor);
    }
}
```

Benefits of subclassing & inheritance

- Don't repeat unchanged fields and methods
  - in implementation:
    - simpler maintenance: fix bugs once (changeability)
  - in specification:
    - clients who understand the superclass specification need only study novel parts of the subclass (readability)
    - differences not buried under mass of similarities
  - modularity: can ignore private fields and methods of superclass (if properly designed)

- Ability to substitute new implementations (modularity)
  - no client code changes required to use new subclasses

Subclassing can be misused

- Poor design can produce subclasses that depend on many implementation details of superclasses
  - super- and sub-classes are often **highly interdependent** (i.e., tightly coupled)
- Changes in superclasses can break subclasses
  - "fragile base class problem"
- Subtyping and implementation inheritance are orthogonal!
  - subclassing gives you both
  - sometimes you want just one. **instead use**:
    - *interfaces*: subtyping without inheritance
    - *composition*: use implementation without subtyping
      - can seem less convenient, but often better long-term

(NON-)EXAMPLES

Is every square a rectangle?

```
interface Rectangle {
  // effects: fits shape to given size:
  //          this_post.width = w, this_post.height = h
  void setSize(int w, int h);
}
interface Square extends Rectangle {…}
```

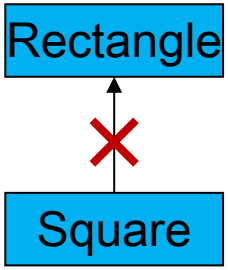
Which is the best option for **`Square`**'s **`setSize`** specification?

```
1.// effects: sets all edges to given size
  void setSize(int edgeLength);
2. // requires: w = h
   // effects: fits shape to given size
  void setSize(int w, int h);
3.// effects:  sets this.width and this.height to w
  void setSize(int w, int h);
4. // effects: fits shape to given size
   // throws BadSizeException if w != h
  void setSize(int w, int h) throws BadSizeException;
```

Square, Rectangle Unrelated (Subtypes)

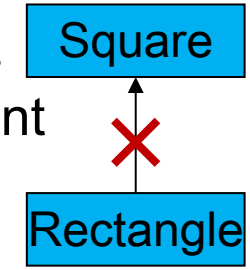
`Square` is not a (true subtype of) **`Rectangle`**:

- **`Rectangle`**s are expected to have a width and height that can be mutated independently
- **`Square`**s violate that expectation, could surprise client



`Rectangle` is not a (true subtype of) **`Square`**:

- **`Square`**s are expected to have equal widths and heights
- **`Rectangle`**s violate that expectation, could surprise client

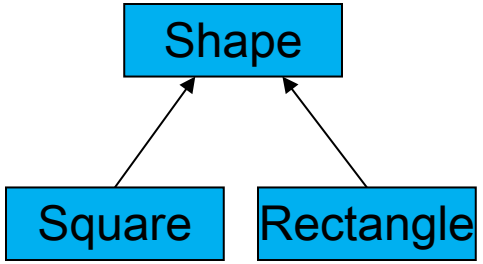


Subtyping is not always intuitive

- but it forces clear thinking and prevents errors

Solutions:

- make them unrelated (or siblings)
- make them immutable!
  - recovers elementary-school intuition



Inappropriate subtyping in the JDK

```
class Hashtable {
  public void put(Object key, Object value){…}
  public Object get(Object key){…}
}

// Keys and values are strings.
class Properties extends Hashtable {
   public void setProperty(String key, String val) {
     put(key,val);
   }
   public String getProperty(String key) {
     return (String)get(key);
   }
}
```

```
Properties p = new Properties();
Hashtable tbl = p;
tbl.put("One", 1);
p.getProperty("One"); // crash!
```

Violation of rep invariant

`Properties` class has a simple rep invariant:

- keys and values are **`String`**s

But client can treat **`Properties`** as a **`Hashtable`**

- can put in arbitrary content, break rep invariant

From Javadoc:

> *Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. ... If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.*

Solution: Composition

```
class Properties {
   private Hashtable hashtable;

   public void setProperty(String key, String value) {
      hashtable.put(key,value);
   }

   public String getProperty(String key) {
      return (String) hashtable.get(key);
   }

   …
}
```

You do not need to be a subclass of every class whose code you want to use!

Now, there are no `get` and `put` methods on `Properties`. (Best choice.)

SUBTYPES VS SUBCLASSES

Substitution principle for methods

Constraints on methods

- For each supertype method, subtype must have such a method
  - (could be inherited or overridden)

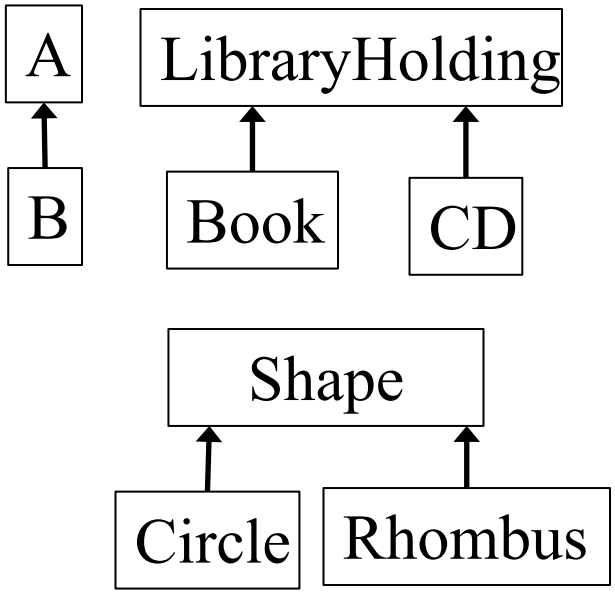
Each overridden method must *strengthen* (or match) the spec:

- ask nothing extra of client ("weaker precondition")
  - *requires* clause is at most as strict as in supertype's method
- guarantee at least as much ("stronger postcondition")
  - *effects* clause is at least as strict as in the supertype method
  - no new entries in *modifies* clause
  - promise more (or the same) in *returns* & *throws* clauses
    - cannot change return values or switch between return and throws

Spec strengthening: argument/result types

For method inputs:

- argument types in A's foo *could* be replaced with supertypes in B's foo
- places no extra demand on the clients
- **but** Java *does not have* such overriding
  - these are different methods in Java!



For method outputs:

- result type of A's foo may be replaced by a subtype in B's foo
- no new exceptions (for values in the domain)
- existing exceptions can be replaced with subtypes (none of this violates what client can rely on)

Recall: Subtyping Example

```
class Product {
    private int price; // in cents
    public int getPrice() {
        return price;
    }
    public int getTax() {
        return (int)(getPrice() * 0.086);
    }
}

class SaleProduct extends Product {
    private float factor;
    public int getPrice() {
      return (int)(super.getPrice()*factor);
    }
}
```

Substitution exercise

Suppose we have a method which, when given one product, recommends another:

```
class Product {
    Product recommend(Product ref);
}
```

Which of these are possible forms of this method in **`SaleProduct`** (a true subtype of **`Product`**)?

```
Product recommend(SaleProduct ref);  // bad

SaleProduct recommend(Product ref);  // good

Product recommend(Object ref); // good, but in Java is
                               overloading

Product recommend(Product ref)  // bad
    throws NoSaleException;
```