CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2022
Subtypes and Subclasses
Set Theory Review

\[(R \setminus S) \setminus T =\]
Set Theory Review

\[(R \setminus S) \setminus T = \text{green} \setminus \text{blue}\]
Set Theory Review

\[(R \setminus S) \setminus T = \text{ green}\]
Set Theory Review

\[(R \setminus S) \setminus T = R \setminus (S \cup T)\]
Set Theory Review

Set theory definition

\[ R \setminus S = R \cap \bar{S} \]

Thus

\[
\begin{align*}
(R \setminus S) \setminus T &= (R \cap \bar{S}) \cap \bar{T} \\
&= R \cap (\bar{S} \cap \bar{T}) \\
&= R \cap (\bar{S} \cup \bar{T}) \\
&= R \setminus (S \cup T)
\end{align*}
\]
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 😞
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 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
- and 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*
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

- Java does not enforce that subclass is a (true) subtype
- 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)
  - “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 {…}

What is wrong with these options 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*:
- *Rectangles* are expected to have a width and height that can be mutated independently
- *Squares* violate that expectation, could surprise client

*Rectangle* is not a (true subtype of) *Square*:
- *Squares* are expected to have equal widths and heights
- *Rectangles* 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
Benefits of Immutability

Seen so far:

1. No worries about **representation exposure**
   – mutable objects need copy-in & copy-out

2. No worries about **equals consistency violations**
   – (no good way to check for this at all!)

3. **Subtyping** relationships more often work as expected
   – e.g., Square is then a subtype of Rectangle
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 StringgetProperty(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 Strings

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 any class whose code you want to use!

Now, there are no get and put methods on Properties. (Best choice.)
SUBTYPES VS SUBCLASS
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);
    }
}

CSE 331 Spring 2022
Substitution exercise

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

```java
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)` throws NoSaleException; // bad