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 by having more expectations than the supertype's specification

We say B is a *true subtype* of A if B has a stronger specification than A
- this is *not* the same as a *Java subtype*
- Java subtypes 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 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
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
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)
    - modularity: can ignore private fields and methods of superclass (if properly designed)
    - readability: differences not buried under mass of similarities

- 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
• 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
Is every square a rectangle?

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

Which is the best option for Square’s setSize specification?
1. // requires: w = h  
   // effects: fits shape to given size  
   void setSize(int w, int h);
2. // effects: sets all edges to given size  
   void setSize(int edgeLength);
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
Inappropriate subtyping in the JDK

class Hashtable<K,V> {
    public void put(K key, V value){...}
    public V get(K key){...}
}

// Keys and values are strings.
class Properties extends Hashtable<Object, Object> {
    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 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 1: Generics

Bad choice:
```java
class Properties extends Hashtable<Object, Object> {
    ...
}
```

Better choice:
```java
class Properties extends Hashtable<String, String> {
    ...
}
```

JDK designers deliberately didn’t do this. Why?
- backward-compatibility (Java didn’t used to have generics)
- (clients can become dependent even on bugs in your code….)
Solution 2: Composition

class Properties {
    private Hashtable<Object, Object> hashtable;

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

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

    ...
}

Now, there are no get and put methods on Properties. (Best choice.)
Substitution principle for classes

If B is a subtype of A, then a B can always be substituted for an A.

Any property guaranteed by A must be guaranteed by B:
- Anything provable about an A is provable about a B.
- If an instance of subtype is treated purely as supertype (only supertype methods/fields used), then the result should be consistent with an object of the supertype being manipulated.

B is permitted to strengthen properties and add properties:
- Fine to add new methods (that preserve invariants).
- An overriding method must have a stronger (or equal) spec.

B is not permitted to weaken a spec:
- No method removal.
- No overriding method with a weaker spec.
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

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)
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`)?

```java
Product recommend(SaleProduct ref); // bad
SaleProduct recommend(Product ref); // OK
Product recommend(Object ref); // OK, but is Java overloading
Product recommend(Product ref) throws NoSaleException; // bad
```
Java subtyping

• Java types:
  – defined by classes, interfaces, primitives

• Java subtyping stems from $B$ extends $A$ and $B$ implements $A$ declarations

• In a Java subtype, each corresponding method has:
  – same argument types
    • if different, then overloading — unrelated methods
  – compatible return types
  – no additional declared exceptions
Java subtyping guarantees

A variable’s run-time type (i.e., the class of its run-time value) is a Java subtype of its declared type.

```
Object o = new Date();  // OK
Date d = new Object();  // compile-time error
```

If a variable of *declared (compile-time)* type $T_1$ holds a reference to an object of *actual (runtime)* type $T_2$, then $T_2$ must be a Java subtype of $T_1$

Corollaries:

- objects always have implementations of the methods specified by their declared type
- *if* all subtypes are true subtypes, then all objects meet the specification of their declared type

Rules out a huge class of bugs
Java subtyping non-guarantees

Java subtyping does **not** guarantee that overridden methods
- have smaller requires
- have smaller modifies
- have stronger postconditions
  - Java only checks the *return type* not the postcondition
  - could compute a completely different function
- have stronger effects
- have stronger throws (& only for the same cases as before)
- have no new unchecked exceptions
Inheritance can break encapsulation

```java
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0; // count # insertions

    public InstrumentedHashSet(Collection<? extends E> c) {
        super(c);
    }

    public boolean add(E o) {
        addCount++;
        return super.add(o);
    }

    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() { return addCount; }
}
```
Dependence on implementation

What does this code print?

```java
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList("CSE", "331"));
System.out.println(s.getAddCount()); // 4?!
```

• Answer **depends on implementation** of `addAll` in `HashSet`
  – different implementations may behave differently!
  – if `HashSet`’s `addAll` calls `add`, then double-counting

• **AbstractCollection**’s `addAll` specification:
  – “adds all elements in the specified collection to this collection.”
  – does not specify whether it calls `add`

• Lesson: subclassing typically requires **designing for inheritance**
  – self-calls is not the only example… (more in future lectures)
Solutions

1. Change spec of `HashSet`
   - indicate all self-calls
   - less flexibility for implementers

2. Avoid spec ambiguity by avoiding self-calls
   a) “re-implement” methods such as `addAll`
      - more work
   b) use composition
      - no longer a subtype (unless an interface is handy)
      - bad for equality tests, callbacks, etc.
Solution 2b: composition

```java
public class InstrumentedHashSet<E> {
    private final HashSet<E> s = new HashSet<E>();
    private int addCount = 0;
    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }
    public boolean add(E o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
    // ... and every other method specified by HashSet<E>
}
```

Delegate

The implementation no longer matters
Composition (wrappers, delegation)

Implementation reuse without inheritance

• Easy to reason about. Self-calls are irrelevant

• Example of a “wrapper” class

• Works around badly-designed / badly-specified classes

• Disadvantages (may be worthwhile price to pay):
  – does not preserve subtyping
  – sometimes tedious to write
  – may be hard to apply to equality tests, callbacks, etc.
    • (although we already saw equals is hard for subclasses)
Composition does not preserve subtyping

• `InstrumentedHashSet` is not a `HashSet` anymore
  – so can't easily substitute it

• It may be a true subtype of `HashSet`
  – but Java doesn't know that!
  – Java requires declared relationships
  – not enough just to meet specification

• Interfaces to the rescue
  – can declare that we implement interface `Set`
  – if such an interface exists
Interfaces reintroduce Java subtyping

```java
public class InstrumentedHashSet<E> implements Set<E> {
    private final Set<E> s = new HashSet<E>();
    private int addCount = 0;
    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }
    public boolean add(E o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    // ... and every other method specified by Set<E>
}
```

What's bad about this constructor?

Avoid encoding implementation details

```
InstrumentedHashSet(Set<E> s) {
    this.s = s;
    addCount = s.size();
}
```
Interfaces and abstract classes

Provide *interfaces* for your functionality
- client code to interfaces rather than concrete classes
- allows different implementations later
- facilitates composition, wrapper classes
  • basis of lots of useful, clever techniques
  • we'll see more of these later

Consider also providing helper/template *abstract classes*
- makes writing new implementations much easier
- not necessary to use them to implement an interface, so retain freedom to create radically different implementations
Java library interface/class example

// root interface of collection hierarchy
interface Collection<E>

// skeletal implementation of Collection<E>
abstract class AbstractCollection<E>
  implements Collection<E>

// type of all ordered collections
interface List<E> extends Collection<E>

// skeletal implementation of List<E>
abstract class AbstractList<E>
  extends AbstractCollection<E>
  implements List<E>

// an old friend...
class ArrayList<E> extends AbstractList<E>
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
Pluses and minuses of inheritance

• Inheritance is a powerful way to achieve code reuse

• Inheritance can break encapsulation
  – a subclass may need to depend on unspecified details of the implementation of its superclass
    • e.g., pattern of self-calls
  – subclass may need to evolve in tandem with superclass
    • okay when implementation of both is under control of the same programmer
  – this is tricky to get right and is a source of subtle bugs

• Effective Java:
  – either design for inheritance or else prohibit it
  – favor composition (and interfaces) to inheritance