Let $P(x)$ be a property provable about objects $x$ of type $T$. Then $P(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.

This means $B$ is a subtype of $A$ if anywhere you can use an $A$, you could also use a $B$.

-- Barbara Liskov

What is subtyping?

Sometimes “every $B$ is an $A”

- Example: In a library database:
  - Every book is a library holding
  - Every CD is a library holding

Subtyping expresses this

- “$B$ is a subtype of $A” means:
  - “every object that satisfies the rules for a $B$ also satisfies the rules for an $A”

Goal: code written using $A$'s specification operates correctly even if given a $B$

- Plus: clarify design, share tests, (sometimes) share code

Subtypes are substitutable

Subtypes are substitutable for supertypes

- 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 that $B$ 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 are confusing and dangerous

- But unfortunately common poor-design 😞
Subtyping vs. subclassing

Substitution (subtype) — a specification notion
- B is a subtype of A if an object of B can masquerade as an object of A in any context
- About satisfiability (behavior of a B is a subset of A’s spec)

Inheritance (subclass) — an implementation notion
- 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() * 0.096);
    }
    ...
}

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

Inheritance makes small extensions small

Much 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
  - In specification
    - Clients who understand the superclass specification need only study novel parts of the subclass
    - Modularity: can ignore private fields and methods of superclass (if properly defined)
    - Differences not buried under mass of similarities
- Ability to substitute new implementations
  - No client code changes required to use new subclasses

Subclassing can be misused

- Poor planning can lead to a muddled class hierarchy
  - Relationships may not match untutored intuition
- Poor design can produce subclasses that depend on many implementation details of superclasses
- 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
  - Interfaces: subtyping without inheritance [see also section]
  - Composition: use implementation without subtyping
    - Can seem less convenient, but often better long-term

Is every square a rectangle?

```java
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 { … }
```

Are any of these good options 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
- Benefit: it forces clear thinking and prevents errors

Solutions:
- Make them unrelated (or siblings)
- Make them immutable (!)
  - Recovers mathematical 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:
class Properties extends Hashtable<Object, Object> {
    ...
}
Better choice:
class Properties extends Hashtable<String, String> {
    ...
}

JDK designers deliberately didn’t do this. Why?
– Backward-compatibility (Java didn’t used to have generics)
– Postpone talking about generics: upcoming lecture

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);
    }
}

...
**Substitution principle for classes**

If B is a subtype of A, 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.

**Spec strengthening: argument/result types**

Method inputs:
- Argument types in A’s foo may be replaced with supertypes in B’s foo (“contravariance”).
- Places no extra demand on the clients.
- But Java does not have such overriding.
  - (Why?)

Method results:
- Result type of A’s foo may be replaced by a subtype in B’s foo (“covariance”).
- 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 principle for methods**

Constraints on methods:
- For each supertype method, subtype must have such a method.
  - Could be inherited or overridden.

Each overriding 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 clause.
  - Throws clause must indicate fewer (or same) possible exception types.

**Substitution exercise**

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

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

Which of these are possible forms of this method in SaleProduct (a true subtype of Product)?
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, overloading: unrelated methods
  - Compatible (covariant) return types
    - A (somewhat) recent language feature, not reflected in (e.g.) clone
    - 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 T1 holds a reference to an object of actual (runtime) type T2, then T2 must be a Java subtype of T1

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

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 of the elements in the specified collection to this collection.”
  - Does not specify whether it calls `add`
- Lesson: Subclassing often requires designing for extension
### Solutions

1. Change spec of `HashSet`
   - Indicate all self-calls
   - Less flexibility for implementers of specification

2. Avoid spec ambiguity by avoiding self-calls
   a) “Re-implement” methods such as `addAll`
      • Requires re-implementing methods
   b) Use a wrapper
      • No longer a subtype (unless an interface is handy)
      • Bad for callbacks, equality tests, 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>
}
```

### 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
  - Tedium to write (your IDE should help you)
  - May be hard to apply to callbacks, equality tests

### 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?

```java
InstrumentedHashSet(Set<E> s) {
    this.s = s;
    addCount = s.size();
}
```

Avoid encoding implementation details

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
- Can minimize number of methods that new implementation must provide
- Makes writing new implementations much easier
- Not necessary to use them to implement an interface, so retain freedom to create radically different implementations that meet an interface

Java library interface/class example

```java
// 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 within a package where implementation of both is under control of same programmer

• Authors of superclass should design and document self-use, to simplify extension
  – Otherwise, avoid implementation inheritance and use composition instead