L14: Inheritance & Composition

Today
- Subclassing & inheritance
- Composition & delegation
- Subtyping

Recommended Reading (from the web)
From David Flanagan, *Java in a Nutshell, 5th edition*
(available through MIT Libraries O’Reilly Safari)
  - Sec 3.5 Subclasses and inheritance (discusses runtime method dispatch and overriding)
From Joshua Bloch, *Effective Java, 2nd edition*
  - Item 16: Favor composition over inheritance
  - Item 17: Design and document for inheritance or else prohibit it

Subclassing
Benefits & risks

**Implementation reuse**

- If class B extends class A, then B inherits A’s fields and method bodies
- This is certainly better than reusing implementation by copying and pasting into B. (Why?)
- But this benefit has a downside: **subclassing breaks encapsulation**
  - A and B can’t be analyzed and changed independently anymore
  - We’ve mentioned this before, and will see more examples today

**Client reuse**

- Java type system allows B objects to be used wherever A objects are used
- So all client code written for A can use B as well
- This benefit also has a downside: **subclasses must be true subtypes**
  - If B objects don’t fulfill the same contract as A objects, then client code will break
  - We’ll see examples later, and define what we mean by subtyping
Subclassing Breaks Encapsulation

Subclassing is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use subclassing within a package, where the subclass and superclass implementation are under the control of the same programmer, and maintained and evolved in tandem. But subclassing in general is not safe, and here's why.

**Subclassing breaks encapsulation.** A subclass depends on the implementation details of its superclass for its proper function. The superclass' implementation may change from release to release, and if it does the subclass may break, even though its code has not been touched. The subclass must evolve in tandem with its superclass, unless the superclass' authors have designed and documented it specifically for the purpose of being extended.

Let's look at a bunch of examples showing what can go wrong with careless subclassing.

**An example from the Java library: java.util.Properties**

```java
class Properties extends Hashtable {
    // Hashtable is an old library class that implements Map<Object,Object>,
    // so Properties inherits methods like:
    Object get(Object key) {...}
    void put(Object key, Object value) {...}

    // Rep invariant of Properties:
    // all keys and values are Strings
    String getProperty(String key) { return (String) this.get(key); }
    void setProperty(String key, String value) { this.put(key, value); }
}
```

**Inherited superclass methods can break the subclass's rep invariant!**

Let’s suppose that we have a program that uses an ArrayList. To tune the performance of our program, we’d like to query the ArrayList as to how many elements have been added since it was created. This is not to be confused with its current size, which goes down when an element is removed. To provide this functionality, we write an ArrayList variant, CountingList, that keeps count of the number of element insertions and exports an observer method for this count. The CountingList class contains two methods capable of adding elements, add and addAll, so we override both of these methods:

```java
class CountingList<E> extends ArrayList<E> {
    private int eltsAdded = 0; // total number of elements ever added

    @Override
    void add(E elt) {
        ++eltsAdded;
        super.add(elt);
    }

    @Override
    void addAll(Collection c) {
        eltsAdded += c.size();
        super.addAll(c);
    }
}
```
What if ArrayList.addAll() works by calling add() \( n \) times?

What if addAll(c) sometimes calls add() \( n \) times, and sometimes does it a different way, depending on the type of the collection? 

When a subclass overrides superclass methods, it may depend on how the superclass uses its own methods.

An example from a photo organizer

// version 1.0 of the photo album class
class Album {
  protected Set<Photo> photos;
  public void addNewPhoto(Photo photo) { photos.add(photo); }
}

// my subclass of the catalog allows removing photos
class MyAlbum extends Album {
  public void removePhoto(Photo photo) { photos.remove(photo); }
}

// version 2.0 of the album comes out
class Album {
  protected Set<Photo> photos;
  protected Map<Person, Photo> photosContaining;
  // Rep invariant: all photos in the index are also found in the photos set
}
// my subclass breaks this new rep invariant

When a class is subclassed, either it must freeze its implementation forever, or all its subclasses must evolve with its implementation.

**Use Composition Rather Than Subclassing**

Instead of extending an existing class, give your new class a private field that references an *instance* of the existing class. This design is called **composition** because the existing class becomes a component of the new one. Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as **forwarding**, or delegation. The resulting class will be solid, with no dependencies on the implementation details of the existing class; the abstraction barrier between the two classes is preserved.

The code below concretizes the approach for Properties and CountingList.

```
class Properties extends Hashtable { ... }
class Properties { private Hashtable table; ... }

class CountingList<E> extends ArrayList<E> { ... }
class CountingList<E> implements List<E> {
  private List<E> list;
  public CountingList<E>(List<E> list) { this.list = list; }

  ... 
} 
```
CountingList is an instance of the **Wrapper** pattern

- A wrapper modifies the behavior of another class without subclassing
- Also decouples wrapper from the specific class it wraps – CountingList could wrap an ArrayList, LinkedList, even another CountingList

**When subclassing is necessary, design for it**

- Define a **protected** API for your subclasses, in the same way you define a **public** API for clients
- Document how you use your own methods (e.g. does addAll() call add()?)
- Don’t expose your rep to your subclasses, or depend on them to maintain your rep invariant

### Subclassing and Equality

Subclassing also causes problems for the definition of equality. Recall our Duration immutable datatype from last lecture:

```java
class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //    mins >= 0, secs >= 0
    // abstraction function:
    //    represents a span of time of mins minutes and secs seconds

    public Duration(int m, int s) {
        mins = m;
        secs = s;
    }

    public long getLength() {
        return mins*60 + secs;
    }

    @Override
    public boolean equals(Object _that) {
        if (!(_that instanceof Duration)) return false;
        Duration that = (Duration) _that;
        return this.getLength() == that.getLength();
    }

    @Override
    public int hashCode() {
        return (int) getLength();
    }
}
```

Suppose we subclass Duration

```java
public class PreciseDuration extends Duration {
    private final int millisecs;
    public PreciseDuration(int m, int s, int ms) {
        super(m,s);
        millisecs = ms;
    }
    public int getMillisecs() {
        return super.getSecs() * 1000 + millisecs;
    }
}
```

How should equality be defined for PreciseDuration?
Can we simply use the equals() inherited from Duration? No, because it ignores milliseconds.

Can we simply override equals() in the same way?

```java
public class PreciseDuration extends Duration {
    ...
    @Override public boolean equals(Object _that) {
        if (!(_that instanceof PreciseDuration)) return false;
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

Not symmetric!

Suppose we use the superclass definition of equals() in all cases except for comparing two PreciseDuration objects:

```java
public class PreciseDuration extends Duration {
    ...
    @Override public boolean equals(Object _that) {
        if (!(_that instanceof PreciseDuration)) {
            return super.equals(_that);
        }
        PreciseDuration that = (PreciseDuration) _that;
        return this.getMillisecs() == that.getMillisecs();
    }
}
```

Not transitive! This allows two different PreciseDuration objects to be equal to the same Duration object.

No really satisfactory solution

Standard approach: superclass equality should reject all subclass objects i.e., instead of

```java
if (!(_that instanceof Duration)) return false;
```

use:

```java
if (!_that.getClass().equals(getClass())) return false;
```

but this is inflexible – e.g., it doesn’t permit a subclass that doesn’t add any new abstract values

Better solution: avoid inheritance, and use composition instead
Subtyping vs. Subclassing

“B is a subtype of A” means “every B object is also an A object”

- Recall that a type is a set of values or objects, so subtyping is simply the subset relation
- In terms of specifications: “every B object satisfies the specification for A”
- **Subclassing (class B extends class A) is one way to declare a subtyping relationship in Java**
- Other ways are implementing an interface (class B implements interface A) or extending an interface (interface B extends interface A)
- When we declare that B is a subtype of A, the Java type system allows us to use a B object wherever an A object is expected
  - i.e., when the declared type of a variable or method parameter is A, the actual runtime type can be B

Substitution Principle

Subtype must be **substitutable** for supertype

- In particular, subtype must fulfill the same contract as the supertype, so that clients designed for the supertype can use subtype objects safely
- Subtype won’t surprise clients by failing to meet guarantees made by the supertype (e.g., postconditions)
- Subtype won’t surprise clients by making stronger demands than the supertype does (e.g. preconditions)

**B is a subtype of A if B’s specification is at least as strong as A’s specification**

- Java compiler guarantees part of this requirement automatically
  - For example, it ensures that every method in A appears in B, with a compatible type signature
- But Java doesn’t check every aspect of the specification
  - Preconditions, postconditions, and other properties are not checked!

If you declare a subtype to Java – e.g. by declaring class B to extend class A – then you should make it substitutable.

Violating the Substitution Principle

Example of failing to provide substitutability:

Rat represents an immutable rational number

```java
class Rat {
    public Rat plus(Rat that);
    public Rat minus(Rat that);
}
```
MutableRat extends Rat to make a mutable rational

```
class MutableRat extends Rat {
    public void addTo(Rat) { ... }
    public void subtractFrom(Rat) { ... }
}
```

This declares to Java that MutableRat is a subtype of Rat… but is MutableRat truly a subtype of Rat?

- Clients that depend on the immutability of Rat may fail when given MutableRat values
  - An immutable expression tree that contains Rat values
  - A function that memoizes previously computed values in HashMap<Rat,Rat>
  - Multithreaded code that uses the same Rat values in different threads

- MutableRat fails to meet guarantees made by Rat
  - Specifically, that the value of the object will never change

- Mutable counterparts of immutable classes should not be declared as subtypes
  - If you want a mutable rational class (e.g. for performance reasons), then it should not be a subtype of Rat
  - String and StringBuffer/StringBuilder offer an example of how to do it right

Another example:

**BigNat represents an immutable natural number (≥ 0)**

```
class BigNat {
    public BigNat plus(BigNat that);
    public BigNat minus(BigNat that);
}
```

**BigInt represents an immutable integer (negatives too)**

```
class BigInt extends BigNat {
    private boolean isNegative;
}
```

BigInt just adds a sign bit to BigNat. Makes sense, right? But is BigInt substitutable for BigNat?

- Abstractly, it doesn’t make any sense
  - Not every integer is a natural!
  - The abstract type of BigInt isn’t a subset of the abstract type of BigNat, so it’s semantic nonsense to declare BigInt a subtype of BigNat

- Practically, it’s risky
  - A function declared to take a BigNat parameter has an implicit precondition that its parameter is ≥ 0
    ```
    double sqrt(BigNat n)
    ```
  - But it can be passed a negative BigInt! What will happen?

- BigInt fails to make guarantees made by BigNat
  - Specifically, that the integer value is not negative
Example #3:

**Immutable Square is a subtype of Immutable Rectangle**

```java
class Rectangle {
    public Rectangle(int w, int h);
    public int getWidth();
    public int getHeight();
}
class Square extends Rectangle {
    public Square(int side) { super(side, side); }
}
```

What about MutableSquare and MutableRectangle?

```java
class MutableRectangle {
    ...
    /** sets this rectangle's dimensions to w x h */
    public void setSize(int w, int h);
}
class MutableSquare extends MutableRectangle {
    ...
    /** requires w = h, sets all edges to given size */
    public void setSize(int w, int h);
    
    Stronger precondition violates MutableRectangle contract.

    /** sets all edges to given size, but throws BadSizeException if w != h */
    void setSize(int w, int h) throws BadSizeException;
    
    Weaker postcondition violates contract, too.

    /** sets all edges to given size */
    void setSize(int side);
    
    Overloads, doesn’t override – so clients can break the rep invariant by calling inherited setSize().
}
```

**Declared subtypes should truly be subtypes**

Design advice:

- When you declare to Java that “B is a subtype of A”, you should ensure that B actually satisfies A’s contract
  - B should guarantee all properties that A does, e.g. immutability
  - B’s methods should have the same or weaker preconditions, and the same or stronger postconditions
- This applies whether the declaration was made using subclassing (class B extends class A) or interface implementation (class B implements interface A) or interface extension (interface B extends interface A)
Inheritance is appropriate only in some circumstances. If you are tempted to have a class B extend a
class A, ask yourself the question: “Is every B really an A?” If you cannot answer yes to this question,
B should not extend A. If the answer is no, it is often the case that B could contain a private instance
of A and expose a smaller and simpler API. A is not an essential part of B, merely a detail of its
implementation.

Even if the answer is yes, you should carefully consider the use of ‘extends’, because as can be seen
by the example of CountingList, the implementation of the subclass may not work due to unspecified
behavior of the superclass. What happens in that example is that the subclass methods break because
the superclass' methods have an implicit dependence between them which is not in the superclass
specification. You should be able to convince yourself that dependences amongst the superclass
methods will not impact subclass behavior before using extends.

**Interfaces vs. Abstract Classes**

Java provides two mechanisms for defining a type that permits multiple implementations: interfaces
and abstract classes. There are two differences between the two mechanisms:

- Abstract classes are permitted to contain implementations for some methods while
  interfaces are not.
- To implement the type defined by an abstract class, a class must be a subclass of the abstract
  class. Any class that defines all of the required methods and obeys the general contract is
  permitted to implement an interface.

Existing classes can be easily retrofitted to implement a new interface. All that needs to be done is to
add the required methods if they don’t yet exist and add an implements clause to the class
declaration. Existing classes cannot, in general, be retrofitted to extend a new abstract class. If you
want to have two classes extend the same abstract class, you have to place the abstract class high up
in the type hierarchy where it subclasses an ancestor of both classes. This might wreak havoc with
the hierarchy. Interfaces allow the construction of non-hierarchical type frameworks. For example,
suppose we have an interface representing a singer and another representing a songwriter:

```java
public interface Singer {
    CD compactdisc(Song s);
}
public interface Songwriter {
    Song compose(boolean hit);
}
```

Some singers could also be songwriters. Because we used interfaces rather than abstract classes to
define these types, it is perfectly permissible for a single class to implement both Singer and
Songwriter. In fact, we can define a third interface that extends both Singer and Songwriter and adds
new methods that are appropriate to the combination:

```java
public interface SingerSongwriter extends Singer, Songwriter {
    void actSensitive();
}
```

You can combine the virtues of interfaces and abstract classes by providing an abstract skeletal
implementation class to go with each nontrivial interface that you export. The interface still defines
the type, but the skeletal implementation takes a lot of the work out of implementing it.

By convention, skeletal implementations are called AbstractInterface, where Interface is the name of
the interface that they implement. For example, the Collections Framework provides a skeletal

---

1 An abstract class is a class that can only be subclassed, and not instantiated.
implementation to go along with each main collection interface: AbstractCollection, AbstractList, etc. The beauty of skeletal implementations is that they provide the implementation assistance of abstract classes without imposing the severe restrictions that abstract classes impose when they serve as type definitions. For most implementors of an interface, extending the skeletal implementation is the obvious choice, but it is strictly optional. If a preexisting class cannot be made to extend the abstract class, then the skeletal implementation can still aid the implementor’s task. The class implementing the interface can forward invocations of interface methods to a contained instance of a private inner class than extends the skeletal implementation. This technique, known as simulated multiple inheritance, is closely related to the wrapper class idiom described above.

Writing a skeletal implementation is a relatively simple, if somewhat tedious, matter. First, you must study the interface and decide which methods are the primitives in terms of which the others can be implemented. These primitives will be the abstract methods in your skeletal implementation. Then you must provide concrete implementations of all the other methods in the interface. For example, here’s a partial skeletal implementation of the Map.Entry interface.

```java
public abstract class AbstractMapEntry implements Map.Entry {
    // Primitives
    public abstract Object getKey();
    public abstract Object getValue();
    ...
    // Implements the general contract of Map.Entry.equals
    @Override public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry arg = (Map.Entry) o;
        return eq(getKey(), arg.getKey()) &&
               eq(getValue(), arg.getValue());
    }
    // Since Object equals was overriden, we better override hashCode!
    @Override public int hashCode() {
        ...
    }
}
```

Summary

Risks of subclassing
- Subclassing breaks encapsulation
- Use subclassing only for true subtyping

Substitution principle
- B is a true subtype of A if and only if B objects are substitutable for A objects anywhere that A is used
- Equivalently, the specification of B must imply the specification of A
- Preconditions of subtype must be weaker, postconditions stronger

Composition
- Wrapper pattern is an alternative to subclassing that preserves encapsulation