Classes and Interfaces

6.170 Lecture 15

Fall 2005

Classes and interfaces lie at the heart of the Java programming language. In this lecture, we describe guidelines to help you design classes and interfaces that are usable, robust and flexible.

This lecture draws from the material in Chapter 4 of Joshua Bloch's book, *Effective Java*.

1 Accessibility

The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules. A well-designed module hides all of its implementation details, cleanly separating its API from its implementation. Modules communicate with each other only through their APIs; further, they don't know each other's inner workings. This concept is termed *information hiding* or *encapsulation*.

The Java programming language has many facilities to aid information hiding. One such facility is the *access control* mechanism, which determines the accessibility of classes, interfaces, and members.

For top-level (non-nested) classes and interfaces, there are only two possible access levels: *package-private* and *public*. If a top-level class or interface can be made package-private, it should be. By making it package-private, you make it part of the package's implementation rather than its exported API, and you can modify it, replace it, or eliminate it in a subsequent release without fear of harming existing clients. If you make it public you are obligated to support it forever to maintain compatibility.

For members (fields, methods, nested classes, and nested interfaces) there are four possible access levels, listed here in order of increasing accessibility:

private : The member is accessible only inside the top-level class where it is declared.

package-private : The member is accessible from any class in the package where it is declared. Technically known as default access, this is the access level you get if no access modifier is specified.

protected : The member is accessible from subclasses of the class where it is declared, subject to a few restrictions that we won't get into, and from any class in the package where it is declared.

public : The member is accessible from anywhere.

The rule of thumb is that you should make each class or member as inaccessible as possible. After carefully designing your class' public API, you should make all other members private, if possible. Both private and package-private members are part of the class implementation and do not normally impact its exported API. For members of public classes, a huge increase in accessibility occurs when the access level goes from package-private to protected. A protected member is part of the class' exported API and must be supported forever.

Note that there is one rule that restricts your ability to reduce the accessibility of methods. If a method overrides a superclass method, it is not permitted to have a lower access level in the subclass than it does in the superclass. This is necessary to ensure that an instance of the subclass is usable anywhere that an instance of the superclass is usable.

2 Dangers of Inheritance

Inheritance (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 inheritance within a package, where the subclass and superclass implementation are under the control of the same programmer.

Unlike method invocation, inheritance 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 suppose that we have a program that uses a `HashSet`. To tune the performance of our program, we need to query the `HashSet` 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 a `HashSet` variant that keeps count of the number of attempted element insertions and exports an accessor for this count. The `HashSet` class contains two methods capable of adding elements, `add` and `addAll`, so we override both of these methods:

```
public class InstrumentedHashSet extends HashSet {
     // The number of attempted element insertions
     private int addCount = 0;

     public InstrumentedHashSet() { }
     public InstrumentedHashSet(Collection c) {
          super(c);
     }

     public boolean add(Object o) {
          addCount++;
          return super.add(o);
     }
     public boolean addAll(Collection c) {
          addCount += c.size();
          return super.addAll(c);
     }
     public int getAddCount() {
          return addCount;
     }
}
```

This class looks reasonable, but it doesn't work. Suppose we create an instance and add three elements using the `addAll` method:

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[] {"S", "Cr", "P"}));
```

We would expect the `getAddCount` method to return three at this point, but it returns six. What went wrong? Internally, `HashSet`'s `addAll` method is implemented on top of its `add` method, although `HashSet` does not document this implementation detail, and it shouldn't have to. The `addAll` method in `InstrumentedHashSet` added three to `addCount` and then invoked `HashSet`'s `addAll` implementation using `super.addAll`. This in turn invoked the `add` method, as overriden in `InstrumentedHashSet`, once for each element. Each of these three invocations added one more to `addCount`, or a total increase of six: Each element added with the `addAll` method is double-counted!

We could "fix" the subclass by eliminating its override of the `addAll` method. While the resulting class would work, it would depend for its proper function on the fact that `HashSet`'s `addAll` method is implemented on top of its `add` method. This "self-use" is an implementation detail, not guaranteed to hold in all implementations of `HashSet` and subject to change from release to release. Therefore, the resulting `InstrumentedHashSet` class would be fragile.

A slightly better solution is to override the `addAll` method to iterate over the specified collection, calling the `add` method once for each element. This amounts to reimplementing superclass methods that may or may not result in self-use, which is difficult and error-prone. Additionally, it is not always possible to do this if the method requires access to private fields inaccessible to the subclass.

The above problem stems from overriding methods. There is a way to avoid this problem using *composition* and *forwarding*.

3 Composition and Forwarding

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. The resulting class will be solid, with no dependencies on the implementation details of the existing class. The code below concretizes the approach.

```
public class CompHashSet {
     private final HashSet s;
     private int addCount = 0;

     public CompHashSet(HashSet s) {
          this.s = s;
     }

     public boolean add(Object o) {
          addCount++;
          return s.add(o);
     }
     public boolean addAll(Collection c) {
          addCount += c.size();
          return s.addAll(c);
     }
     public int getAddCount() {
          return addCount;
     }
}
```

Now, in order to get a `CompHashSet` we first need to create a new `HashSet` and pass it to the constructor for `CompHashSet`. The `CompHashSet` class is known as a *wrapper* class because each `CompHashSet` instance wraps another `HashSet` instance. This wrapping technique is very powerful – we briefly mentioned how wrapping can be used to replace a type's `equals` method by a more suitable one in Lecture 13.

The main disadvantage of wrapper classes is that it is a bit tedious to write forwarding methods, but the tedium is partially offset by the fact that you only have to write one constructor.

4 Inheritance versus Composition/Forwarding

Inheritance is appropriate only in circumstances where the subclass really is a *subtype* of the superclass, as we defined in Lecture 14. In other words, a class B should extend a class A if B is a subtype of A. 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 should 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. In the example above `InstrumentedHashSet` is not a true subtype of `HashSet` because there are additional modified variables, namely `addCount`.

Even if the answer is yes, you should carefully consider the use of 'extends', because as can be seen by the example of `InstrumentedHashSet`, 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.

There are a number of violations of this principle in the Java platform libraries. For example, a stack is not a vector, so `Stack` should not extend `Vector`. Similarly, a property list is not a hash table, so `Properties` should not extend `Hashtable`. In both cases, composition would be appropriate.

Using inheritance where composition is appropriate can lead to confusing semantics. For example, if `p` refers to a `Properties` instance, then `p.getProperty(key)` may yield different results from `p.get(key)`. The former method takes defaults into account, while the latter method, which is inherited from `Hashtable`, does not. More seriously, the client may be able to corrupt invariants of the subclass by modifying the superclass directly. In the case of `Properties`, the designers intended that only strings be allowed as keys and values, but direct access to the underlying `Hashtable` allows this invariant to be violated. Once this invariant is violated, it is no longer possible to use other parts of the `Properties` API (`load` and `store`). By the time the problem was discovered, it was too late to correct it because clients depended on the use of nonstring keys and values.

5 Interfaces versus Abstract Classes

Java provides two mechanisms for defining a type that permits multiple implementations: interfaces and abstract classes[1]. 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 requires 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:

```
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:

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

[1]An abstract class is a class that can only be subclassed, but not instantiated.

6 Skeletal Implementations

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 `Abstract`*Interface*, where *Interface* is the name of the interface that they implement. For example, the Collections Framework provides a skeletal 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.

```
public abstract class AbstractMapEntry implements Map.Entry {
    // Primitives
    public abstract Object getKey();
    public abstract Object getValue();
       ...

    // Implements the general contract of Map.Entry.equals
    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!
    public int hashCode() {
         return
             (getKey() == null ? 0: getKey().hashCode()) ^
             (getValue() == null ? 0: getValue().hashCode());
    }
}
```