Item 17: Design and document for inheritance or else prohibit it

Item 16 alerted you to the dangers of subclassing a "foreign" class that was not designed and documented for inheritance. So what does it mean for a class to be designed and documented for inheritance?

First, the class must document precisely the effects of overriding any method. In other words, the class must document its self-use of overridable methods. For each public or protected method or constructor, the documentation must indicate which overridable methods the method or constructor invokes, in what sequence, and how the results of each invocation affect subsequent processing. (By overridable, we mean nonfinal and either public or protected.) More generally, a class must document any circumstances under which it might invoke an overridable method. For example, invocations might come from background threads or static initializers.

By convention, a method that invokes overridable methods contains a description of these invocations at the end of its documentation comment. The description begins with the phrase "This implementation." This phrase should not be taken to indicate that the behavior may change from release to release. It connotes that the description concerns the inner workings of the method. Here's an example, copied from the specification for java.util.AbstractCollection:

```java
public boolean remove(Object o)

Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element e such that (o==null ? e==null : o.equals(e)), if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).

This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection's iterator method does not implement the remove method.

This documentation leaves no doubt that overriding the iterator method will affect the behavior of the remove method. Furthermore, it describes exactly how the behavior of the Iterator returned by the iterator method will affect the behavior of the remove method. Contrast this to the situation in Item 16, where the
programmer subclassing HashSet simply could not say whether overriding the add method would affect the behavior of the addAll method.

But doesn’t this violate the dictum that good API documentation should describe what a given method does and not how it does it? Yes, it does! This is an unfortunate consequence of the fact that inheritance violates encapsulation. To document a class so that it can be safely subclassed, you must describe implementation details that should otherwise be left unspecified.

Design for inheritance involves more than just documenting patterns of self-use. To allow programmers to write efficient subclasses without undue pain, a class may have to provide hooks into its internal workings in the form of judiciously chosen protected methods or, in rare instances, protected fields. For example, consider the removeRange method from java.util.AbstractList:

```java
protected void removeRange(int fromIndex, int toIndex)
```

Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the AbstractList by (toIndex - fromIndex) elements. (If toIndex == fromIndex, this operation has no effect.)

This method is called by the clear operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the clear operation on this list and its sublists.

This implementation gets a list iterator positioned before fromIndex and repeatedly calls ListIterator.next followed by ListIterator.remove, until the entire range has been removed. Note: If ListIterator.remove requires linear time, this implementation requires quadratic time.

Parameters:
- fromIndex index of first element to be removed.
- toIndex index after last element to be removed.

This method is of no interest to end users of a List implementation. It is provided solely to make it easy for subclasses to provide a fast clear method on sublists. In the absence of the removeRange method, subclasses would have to make do with quadratic performance when the clear method was invoked on sublists or rewrite the entire subList mechanism from scratch—not an easy task!
So how do you decide what protected members to expose when you design a class for inheritance? Unfortunately, there is no magic bullet. The best you can do is to think hard, take your best guess, and then test it by writing subclasses. You should expose as few protected members as possible, because each one represents a commitment to an implementation detail. On the other hand, you must not expose too few, as a missing protected member can render a class practically unusable for inheritance.

The only way to test a class designed for inheritance is to write subclasses. If you omit a crucial protected member, trying to write a subclass will make the omission painfully obvious. Conversely, if several subclasses are written and none uses a protected member, you should probably make it private. Experience shows that three subclasses are usually sufficient to test an extendable class. One or more of these subclasses should be written by someone other than the superclass author.

When you design for inheritance a class that is likely to achieve wide use, realize that you are committing forever to the self-use patterns that you document and to the implementation decisions implicit in its protected methods and fields. These commitments can make it difficult or impossible to improve the performance or functionality of the class in a subsequent release. Therefore, you must test your class by writing subclasses before you release it.

Also, note that the special documentation required for inheritance clutters up normal documentation, which is designed for programmers who create instances of your class and invoke methods on them. As of this writing, there is little in the way of tools or commenting conventions to separate ordinary API documentation from information of interest only to programmers implementing subclasses.

There are a few more restrictions that a class must obey to allow inheritance. Constructors must not invoke overridable methods, directly or indirectly. If you violate this rule, program failure will result. The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run. If the overriding method depends on any initialization performed by the subclass constructor, the method will not behave as expected. To make this concrete, here's a class that violates this rule:

```java
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }
    public void overrideMe() {
    }
}
```
Here's a subclass that overrides the `overrideMe`, method which is erroneously invoked by Super's sole constructor:

```java
public final class Sub extends Super {
    private final Date date; // Blank final, set by constructor
    Sub() {
        date = new Date();
    }

    // Overriding method invoked by superclass constructor
    @Override public void overrideMe() {
        System.out.println(date);
    }

    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

You might expect this program to print out the date twice, but it prints out `null` the first time, because the `overrideMe` method is invoked by the `Super` constructor before the `Sub` constructor has a chance to initialize the date field. Note that this program observes a final field in two different states! Note also that if `overrideMe` had invoked any method on date, the invocation would have thrown a `NullPointerException` when the `Sub` constructor invoked `overrideMe`. The only reason this program doesn't throw a `NullPointerException` as it stands is that the `println` method has special provisions for dealing with a null argument.

The `Cloneable` and `Serializable` interfaces present special difficulties when designing for inheritance. It is generally not a good idea for a class designed for inheritance to implement either of these interfaces, as they place a substantial burden on programmers who extend the class. There are, however, special actions that you can take to allow subclasses to implement these interfaces without mandating that they do so. These actions are described in Item 11 and Item 74.

If you do decide to implement `Cloneable` or `Serializable` in a class designed for inheritance, you should be aware that because the `clone` and `readObject` methods behave a lot like constructors, a similar restriction applies: **neither `clone` nor `readObject` may invoke an overridable method, directly or indirectly.** In the case of the `readObject` method, the overriding method will run before the subclass's state has been deserialized. In the case of the `clone` method, the overriding method will run before the subclass's `clone` method has a chance to
fix the clone’s state. In either case, a program failure is likely to follow. In the case of clone, the failure can damage the original object as well as the clone. This can happen, for example, if the overriding method assumes it is modifying the clone’s copy of the object’s deep structure, but the copy hasn’t been made yet.

Finally, if you decide to implement Serializable in a class designed for inheritance and the class has a readResolve or writeReplace method, you must make the readResolve or writeReplace method protected rather than private. If these methods are private, they will be silently ignored by subclasses. This is one more case where an implementation detail becomes part of a class’s API to permit inheritance.

By now it should be apparent that designing a class for inheritance places substantial limitations on the class. This is not a decision to be undertaken lightly. There are some situations where it is clearly the right thing to do, such as abstract classes, including skeletal implementations of interfaces (Item 18). There are other situations where it is clearly the wrong thing to do, such as immutable classes (Item 15).

But what about ordinary concrete classes? Traditionally, they are neither final nor designed and documented for subclassing, but this state of affairs is dangerous. Each time a change is made in such a class, there is a chance that client classes that extend the class will break. This is not just a theoretical problem. It is not uncommon to receive subclassing-related bug reports after modifying the internals of a nonfinal concrete class that was not designed and documented for inheritance.

The best solution to this problem is to prohibit subclassing in classes that are not designed and documented to be safely subclassed. There are two ways to prohibit subclassing. The easier of the two is to declare the class final. The alternative is to make all the constructors private or package-private and to add public static factories in place of the constructors. This alternative, which provides the flexibility to use subclasses internally, is discussed in Item 15. Either approach is acceptable.

This advice may be somewhat controversial, as many programmers have grown accustomed to subclassing ordinary concrete classes to add facilities such as instrumentation, notification, and synchronization or to limit functionality. If a class implements some interface that captures its essence, such as Set, List, or Map, then you should feel no compunction about prohibiting subclassing. The wrapper class pattern, described in Item 16, provides a superior alternative to inheritance for augmenting the functionality.
If a concrete class does not implement a standard interface, then you may inconvenience some programmers by prohibiting inheritance. If you feel that you must allow inheritance from such a class, one reasonable approach is to ensure that the class never invokes any of its overridable methods and to document this fact. In other words, eliminate the class’s self-use of overridable methods entirely. In doing so, you’ll create a class that is reasonably safe to subclass. Overriding a method will never affect the behavior of any other method.

You can eliminate a class’s self-use of overridable methods mechanically, without changing its behavior. Move the body of each overridable method to a private “helper method” and have each overridable method invoke its private helper method. Then replace each self-use of an overridable method with a direct invocation of the overridable method’s private helper method.
Item 18: Prefer interfaces to abstract classes

The Java programming language provides two mechanisms for defining a type that permits multiple implementations: interfaces and abstract classes. The most obvious difference between the two mechanisms is that abstract classes are permitted to contain implementations for some methods while interfaces are not. A more important difference is that 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, regardless of where the class resides in the class hierarchy. Because Java permits only single inheritance, this restriction on abstract classes severely constrains their use as type definitions.

Existing classes can be easily retrofitted to implement a new interface. All you have to do is add the required methods if they don’t yet exist and add an implements clause to the class declaration. For example, many existing classes were retrofitted to implement the Comparable interface when it was introduced into the platform. 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. Unfortunately, this causes great collateral damage to the type hierarchy, forcing all descendants of the common ancestor to extend the new abstract class whether or not it is appropriate for them to do so.

Interfaces are ideal for defining mixins. Loosely speaking, a mixin is a type that a class can implement in addition to its “primary type” to declare that it provides some optional behavior. For example, Comparable is a mixin interface that allows a class to declare that its instances are ordered with respect to other mutually comparable objects. Such an interface is called a mixin because it allows the optional functionality to be “mixed in” to the type’s primary functionality. Abstract classes can’t be used to define mixins for the same reason that they can’t be retrofitted onto existing classes: a class cannot have more than one parent, and there is no reasonable place in the class hierarchy to insert a mixin.

Interfaces allow the construction of nonhierarchical type frameworks. Type hierarchies are great for organizing some things, but other things don’t fall neatly into a rigid hierarchy. For example, suppose we have an interface representing a singer and another representing a songwriter:

```java
public interface Singer {
    AudioClip sing(Song s);
}
```