So how do you decide what protected methods or fields to expose when designing 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 some subclasses. You should provide as few protected methods and fields as possible because each one represents a commitment to an implementation detail. On the other hand, you must not provide too few, as a missing protected method can render a class practically unusable for inheritance.

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.

Also, note that the special documentation required for inheritance clutters up the 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 this rule is violated, it is likely that 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, then the method will not behave as expected. To make this concrete, here's a tiny class that violates this rule:

```
public class Super {
    // Broken - constructor invokes overridable method
    public Super() {
        m();
    }

    public void m() {
    }
}
```


Here's a subclass that overrides `m`, which is erroneously invoked by `Super`'s sole constructor:

```
final class Sub extends Super {
    private final Date date; // Blank final, set by constructor

    Sub() {
        date = new Date();
    }

    // Overrides Super.m, invoked by the constructor Super()
    public void m() {
        System.out.println(date);
    }

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

You might expect this program to print out the date twice, but it prints out `null` the first time because the method `m` is invoked by the constructor `Super()` before the constructor `Sub()` has a chance to initialize the `date` field. Note that this program observes a final field in two different states.

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 10 and Item 54.

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` methods has a chance to fix the clone's state. In either case, a program failure is likely to follow. In the case of the `clone` method, the failure can do damage to the object being cloned as well as to the clone itself.