Figure 16.3 Abstract classes.

Our **Mark** class hierarchy looks like this:

**Mark** has an **intValue()** method that is overridden in both **LetterGrade** and **PercentGrade**. In fact, it *must be* overridden because we don't know how to implement it in **Mark**:

```
public class Mark {
    public int intValue () {
        return -9999;   // Java insists we return something
    }
}
```

We would like to just leave **intValue()** out of **Mark**, but we can't. Why not? What we need instead is a new method classification that carries two meanings:

1. This method is required in any actual object, and
2. we can't say how to implement it in general.

In Java, a method that satisfies these two criteria is an *abstract method* and its class is an *abstract class*:

```
abstract public class Mark {
    abstract public int intValue ();    // note the semicolon

    public double gradePoint () {
        int m = this.intValue();
        if (m >= 85) return 4.00;
        else if (m >= 80) return 3.70;
        …
    }
}
```

Here are some rules about abstract methods and classes:

1. If any method is abstract, the class must be abstract.
2. An abstract class may have non-abstract methods.
3. An abstract class *cannot be instantiated*, but may have constructors that can be called by its subclass's constructors.
4. Subclasses of an abstract class that wish to be concrete (i.e. non-abstract) must override *all* inherited abstract methods with full implementations.