A Student Grading Example

Specification:

Maintain a class roster for a class containing both undergraduate and graduate students. Each student has a name, three test scores, and a final grade.

Undergraduate students final grade is:
    Pass if \((test1+test2+test3)/3 \geq 70\)

Graduate students final grade is:
    Pass if \((test1+test2+test3)/3 \geq 80\)

We will assume that an array of students exists and just calculate the final grades for these students.
Procedural Solution:

```java
public class Student {
    public String name;      // name of student
    public boolean graduate; // true for graduate
    public int t1, t2, t3;   // test scores
    public boolean pass;     // true if passed
} // Student

public class Example {
    public static void main( String args [] ) {
        Student roster[] = new Student[100];
        // initialize roster array
        for (int i = 0; i < roster.length; i++) {
            if ( ! roster[i].graduate ) {
                roster[i].pass = (t1+t2+t3)/3 >= 70;
            } else {
                roster[i].pass = (t1+t2+t3)/3 >= 80;
            }
        }
    }
} // Example
```
Poor Object-Oriented Solution:

```java
public class Student {
    private String name;      // name of student
    private boolean graduate; // true if graduate
    private int t1, t2, t3;   // test scores
    private boolean pass;     // true if passed

    public void setTest1( int grade ) {
        t1 = grade;
    }
    ... (more setters) ...
    public boolean getPass() {
        return pass;
    }
    ... (more getters) ...
    public void calculateGrade() {
        pass = ( t1 + t2 + t3 ) / 3
             >= ( graduate ? 80 : 70 );
    }
} // Student

public class Example {
    public static void main( String args [] ) {
        Student roster [] = new Student[100];
        // initialize roster array
        for (int i = 0; i < roster.length; i++) {
            calculateGrade( roster[i] );
        }
    }
} // Example
```
Polymorphic Solution:
public class Student {
    private String name;      // name of student
    private int t1, t2, t3;   // test scores
    private boolean pass;     // true if passed
    ... // need to define calculateGrade here
} // Student

public class GraduateStudent extends Student {
    public void calculateGrade() {
        pass = ( t1 + t2 + t3 ) / 3 >= 80;
    }
} // GraduateStudent

public class UndergraduateStudent
    extends Student {
    public void calculateGrade() {
        pass = ( t1 + t2 + t3 ) / 3 >= 70;
    }
} // UndergraduateStudent

public class Example {
    public static void main( String args [] ) {
        Student roster[] = new Student[100];
        // initialize roster array
        for (int i = 0; i < roster.length; i++) {
            roster[i].calculateGrade();
        }
    }
} // Example
Polymorphism

Polymorphism is the ability to choose a method to execute based on the actual type of an object.

In Java, the type of the variable referring to an object must define all methods invoked using the variable.

The actual method invoked will depend on the actual object the variable refers to.

This means that all methods you want to call on a superclass variable must be defined in the superclass.

Polymorphism only works for methods, not for fields (instance variables) - that is why it is better to access instance variables through accessor methods.

Abstract classes are used when a method must be defined in a superclass but an actual implementation for the method makes no sense there.
When to use inheritance

If all the answers are yes you should consider using inheritance

  Do all instances of the subclass have all the behavior and state as instances of the superclass?
  Do the instances of the subclass have additional or slightly different behavior from that of instances of the superclass?
  Is all of the behavior and state of the superclass appropriate for the subclass?

If an entity A is a specialized form of another entity B, that is, A is a B, then model them with inheritance. Declare A as a subclass of B.
Interfaces

Sometimes we would like an object to inherit from multiple classes. This is called “multiple inheritance”.

Java does not support multiple inheritance.

Java does define “interfaces” to allow objects to be declared to implement multiple sets of methods.

A programmer can then declare a class to “implement” several interfaces. This allows objects of the class to be referred to by variables of the interface type and methods can be invoked on the variable that are declared in the interface.

A class that implements an interface does not have to be in any special super/sub class relationship with the interface.
Interface declarations

Generally one interface declaration per file:

```
modifiers interface InterfaceName {
    interface body declarations ...
}
```

Interface modifiers

Top-level interfaces cannot be protected or private.
All interfaces are abstract so this modifier is redundant and so as a matter of style should not be used.
Interfaces cannot be final

interface body declarations:

field declarations:
```
public static final Type variables;
```
All fields are public static final and must be constant declarations.

constructors:
Constructors are not allowed in interfaces

method declarations:
```
abstract public returnType name ( optional arguments ) ;
```
All methods in an interface are public and abstract. As a matter of style these modifiers are not used in interface declarations as they are assumed.

...
Using interfaces

To use an interface name the interfaces after the implements keyword:

    modifiers class ClassName implements Interfaces {
        class body declarations ...
    }

All methods declared in the interface(s) must be implemented in the class (or the class must be declared abstract).
Interfaces

The shape interface might be useful in other contexts

There might be a furniture store selling different kinds of tables. Tables are arranged in a hierarchy with the superclass “Table” with subclasses “CoffeeTable” and “KitchenTable”.

Tables have a name, perimeter, and area.

If we define “Shape” as an interface we can then implement this interface in the Table classes.

```java
public class CoffeeTable
    extends Table
    implements Shape {

}
Classes, Subclasses, interfaces, abstract classes

A class is declared with a class declaration

If there is no extends keyword then the class is a direct subclass of class Object.

A subclass is declared by specifying a superclass after the extends keyword

final classes cannot have subclasses

An abstract class is declared like a class but with the modifier abstract

An interface is declared with an interface declaration

A subinterface is declared by specifying a superinterface after the extends keyword

A subclass of an abstract class must implement all abstract methods or be declared abstract.

A class that implements superinterfaces must implement all methods declared in the interface(s) or be declared abstract.
The `instanceof` operator

If we have a variable `s` of type `Shape` that refers to an object of type `Circle` we cannot access any properties of the `Circle` that are not defined for `Shapes`.

We can write:

```java
Circle c = (Circle) s;
```

and cast the type and then use all the `Circle` methods on the variable `c`.

If `s` refers to a `Rectangle` then the cast will throw an exception (and very likely abort the program).

We can test if the cast will work with the `instanceof` operator:

```java
"s instanceof Circle"
```

This expression is `true` if the object referred to by `s` is an instance of a `Circle` and `false` otherwise.

```java
if (s instanceof Circle) {
    Circle c = (Circle) s;
    // use c as a Circle reference
} else {
    // do something if s is not a circle
}
```