Declaring Interfaces

- What is an interface?
  - A set of method and constant declarations, without the method implementations.
    * Example

```
public interface Colorable {
  public void setColor(int color);
  public int getColor();
}
```

  - One interface can *extend* another interface.
    * Example

```
public interface Paintable extends Colorable {
  public static final int MATTE = 0, GLOSSY = 1;
  public void setFinish(int finish);
  public int getFinish();
}
```

Using Interfaces

- How do you use an interface?
  - In a class definition, we say that a class *implements* an interface.
    - Example

```
class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
  int _color;
  public void setColor(int color) { _color = color; }
  public int getColor() { return _color; }
}
```

  - An interface is a reference type, just like a class.
    - Example

```
Colorable widget = new ColoredPoint();
widget.setColor(GREEN);
```

Using Interfaces (cont.)

- A class can implement one or *more* interfaces.
  - Example #1

```
class MyClass implements IYourInterface1,
                         IYourInterface2 {
  . . .
}
```

  - Example #2

```
class PaintedPoint extends ColoredPoint implements Paintable
{
  int _finish;
  public void setFinish(int finish) {
    _finish = finish;
  }
  public int getFinish() { return _finish; }
}
```

The Standard Visitor Pattern

- Our definition of the Visitor Pattern differs from the (standard) definition presented by the GoF book ("Design Patterns").
  - For example, they would define the methods of `IListAlgo` as follows:
    1. `Object forEmpty(EmptyList host, Object input)` to act on `EmptyList` objects and
    2. `Object forNonEmpty(NEList host, Object input)` to act on `NEList` objects only.
       * In contrast, we defined the `host` as an `AList` for both methods.

Software Engineering Issues

- It is good software engineering practice to shield clients from the details of correctly manufacturing concrete instances of a list.
  - For this reason, the constructors for `EmptyList` and `NEList` are package private.
  - A factory class, `ListFactory`, is provided to build `EmptyList` and `NEList` objects.
    - It checks for valid input before calling on the appropriate constructors to instantiate and initialize concrete list objects.
    - `ListFactory` resides in the same package as `EmptyList` and `NEList` and thus can access all package private elements.

Software Engineering Issues (cont.)

- Each of the visitor's methods explicitly prescribes what concrete subclass of `AList` must be passed to it as a parameter.
  - As a consequence, `AList` and all of its subclasses must be public in order for any concrete visitor to use them.
    - In practice, the developer of this list/visitor framework would deliver `AList`, `EmptyList`, `NEList`, `IListAlgo`, and `ListFactory` in one package to the client.
    - Any client can develop any concrete visitors to add on to the existing system without rewriting/recompiling any of the existing code.
      - The concrete visitors are usually in different packages created by the clients to suit their needs.
      - Since `AList`, `EmptyList`, `NEList`, `IListAlgo`, and `ListFactory` are all public classes, they can be directly manipulated by any client via their public behaviors.

Software Engineering Issues (cont.)

- It is good software engineering practice to program at the highest level of abstraction (OOPP#2: *Program to the (abstract) interface*).

- In the preceding version of the polynomial/visitor framework, the visitor interface requires a specific concrete subclass of `AList` for each of its methods and thus violates this principle.

Software Engineering Issues (cont.)

- We would like to hide more of the details of the implementation from the clients: `EmptyList` and `NEList` should be hidden from the clients and made package private.
  - This will allow us more flexibility in modifying our implementation of `AList` without changing any of the clients' code.
  - We can achieve this goal because in our current design, `EmptyList` and `NEList` have the same public methods as their abstract superclass `AList`.
    * And since the visitors only deal with the public methods of the host, they need not know about the concrete subclasses of `AList`.
    * We can promote the standard visitor pattern to a higher level of abstraction by making the visitor interface depend only on the abstract host.

A Variant of the Visitor Pattern

- The only change we need to make is to redefine the visitor interface `IListAlgo` and the corresponding method signatures of all of its concrete implementations to require `AList` as a host instead:

  1. `Object forEmpty(AList host, Object input)` to act on `EmptyList` objects only, and
  2. `Object forNonEmpty(AList host, Object input)` to act on `NEList` objects only.

- Everything else remains the same.

  – Polymorphism will ensure that, at run time, the proper calls will be made by the proper concrete subclass, reducing code complexity.

A Variant of the Visitor Pattern (cont.)

- By hiding the details of implementation and exposing only the abstract class `AList` to all of its clients, in particular its visitors, we can change the implementation of `AList` without affecting any of the existing client code.