Understanding `Private` and `Protected`

- Can one object access the private and protected members (i.e., fields, methods, and constructors) of another object of the same class?
  - For example, is the following class definition allowed?

```
class MyInteger {
  private int iAmPrivate;

  boolean isEqualTo(MyInteger anotherInteger) {
    return iAmPrivate == anotherInteger.iAmPrivate;
  }
}
```

Understanding `Private` and `Protected` (cont.)

- *Yes.*
  - Objects of the same type have access to one another's private and protected members.
    - This is because access restrictions apply at the class or type level (all instances of a class) rather than at the object level (this particular instance of a class).

Understanding `Private` and `Protected` (cont.)

- Example

```
class ConstPoly extends APolynomial
{
  ...
  // If parameter APolynomial p is a constant polynomial,
  // returns a ConstPoly whose coefficient is the sum
  // of the coefficients of this ConstPoly and p.  Otherwise,
  // asks p to add this APolynomial to itself.
  public APolynomial add(APolynomial p)
  {
    return (0 == p.getDegree()) ? new ConstPoly(_coef +
                                                p._coef)
                                : p.add(this);
  }
```

"this"

- In any method or constructor, `this` refers to the object on which the method is being performed. It's useful when you ...

  1. need to access a field that is obscured by a parameter or
  2. want to pass the object as an argument to a method.

What's Wrong With This Picture?

- Each time we want to compute something new, we edit each class in order to add methods to it.

- Is there a way to add new behavior to `AList` or `APolynomial` without touching any of the existing code, leaving everything that has been written so far unchanged?

Toward a Solution ...

- The key is to encapsulate the variant behaviors in a separate Union Pattern (OOPP #1).
  - The *invariant* behaviors are the constructor and methods `getFirst()` and `getRest()`.
  - The *variant* behaviors are the infinitely many algorithms (i.e. computations) that we want `AList` to perform.
    * For `AList` to execute any of these algorithms, we just need to add one more method to `AList`.

The Visitor Pattern

- The visitor pattern is a framework for communication and collaboration between two union patterns: a "host" union and a "visitor" union.
  - An abstract visitor is usually defined as an interface in Java.
    - It has a separate method for each of the concrete variants of the host union.

```
public interface IListAlgo {

  public abstract Object forEmpty(AList host,
                                  Object input);

  public abstract Object forNonEmpty(AList host,
                                     Object input);
}
```

– The abstract host has a method (called the "hook") to "accept" a visitor ...

```
public abstract class AList
{
  ...
  public abstract Object execute(IListAlgo algo,
                                 Object input);
  ...
}
```

– and leaves it up to each of its concrete variants to call the appropriate visitor method.

```
class NEList extends AList {
  ...
  public Object execute(IListAlgo algo, Object input)
  {
    return algo.forNonEmpty(this, input);
  }
  ...
}
```

The Visitor Pattern (cont.)

- The concrete visitor implements the interface defined by the abstract visitor.

```
public class Length implements IListAlgo
{
  public final static Length Singleton = new Length();

  private Length()
  {
  }

  public Object forEmpty(AList host, Object input)
  {
    ...
  }
```

```
    public Object forNonEmpty(AList host, Object input)
    {
       ...
    }
  }
```

The Visitor Pattern (cont.)

- This "decoupling" of the host's structural behaviors from the extrinsic algorithms on the host permits the addition of infinitely many external algorithms without changing any of the host union code.

- This extensibility only works if the taxonomy of the host union is stable and does not change.
  - If we have to modify the host union, then we will have to modify ALL visitors as well!

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; }
}
```