Overview

- The Visitor Pattern
- Interfaces

What's The Matter?

- Each time we want to compute something new, we have to edit each class and add appropriate methods to each class.

- Is there a way to add new behavior to `List` 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 into a separate Union Pattern (OOPP #1).
  - Here, the variant behaviors are the infinitely many algorithms (i.e. computations) that we want List to perform.
  - The invariant behaviors are the methods `find()`, `insert()`, and `remove()`.
  - For List to execute any of these algorithms, we just need to add to the design of List one more method and never have to modify anything ever again!

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.
  - The abstract host has a method (called the "hook") to "accept" a visitor and leaves it up to each of its concrete variants to call the appropriate visitor method.
    - 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!

The Visitor Pattern (cont.)

- In practice, the host union is encapsulated inside of another class, say `Structure`.
  - A client program, say `StructClient`, only deals with the `Structure` class and the `IVisitor` interface.
  - An appropriate "factory" will provide the client with concrete visitors that it wants.
    - All the "state-less" visitors should be singletons and are factories that manufacture unique instances of themselves!

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; }
}
```