Code Reuse

Composition

Code Reuse

- Effective software development relies on reusing existing code. Why?
  - Save time
- Code reuse is more than copying code and adapting it
  - Typical approach in procedural languages like C
- Code reuse in C?
  - Call one function inside another
- One goal in OOP
  - Reuse code (classes) without changing the classes
- Code reuse in Java?
  - Invoke one method (same class or not) inside another method
  - Build classes (state + functionality) based on another classes
    - OOP technique known as **Composition**
  - Inheritance (later)

Composition

- Composition
  - OOP technique to reuse code
  - Design method characterized by defining the **state** and **behavior** of instances based on instance variables of other classes
  - Build one class using other classes
- Also known as **aggregation**

- Also referred to as a *has-a* relationship

Rectangle Example

Rectangle without Composition

```
public class Rectangle{
  private int _originX, _originY;
  private int _endX, _endY;

  public Rectangle(int x, int y,
                   int ex, int ey) {
    _originX = x; _originY = y;
    _endX = ex; _endY = ey;
  }

  public void move(int dx, int dy) {
    _originX += dx;
    _originY += dy;
    _endX += dx;
    _endY += dy;
  }

  public boolean equals(Rectangle r) {
   return (r != null) &&
          _originX == r._originX &&
          _originY == r._originY &&
          _endX == r._endX &&
          _endY == r._endY;
  }
}
```

Rectangle with Composition – Reusing Point

```
public class Rectangle{
  private Point _origin;
  private Point _end;

  public Rectangle(int x, int y,
                   int ex, int ey) {
    _origin = new Point(x, y);
    _end = new Point(ex, ey);
  }
  public Rectangle(Point o, Point e) {
    _origin = o;
    _end = e
  }

  public void move(int dx, int dy) {
    _origin.move(dx, dy);
    _end.move(dx, dy);
  }

  public boolean equals(Rectangle r) {
   return (r != null) &&
          _origin.equals(r._origin) &&
          _end.equals(r._end);
  }
}
```

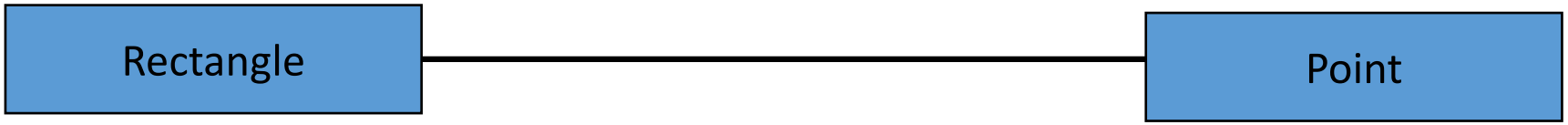
Designing Rectangle Class

- Code is reused in two methods of `Rectangle`
  - `move`
    - Invoke (twice) `move` method of `Point`
  - `equals`
    - Invoke (twice) `equals` method of `Point`

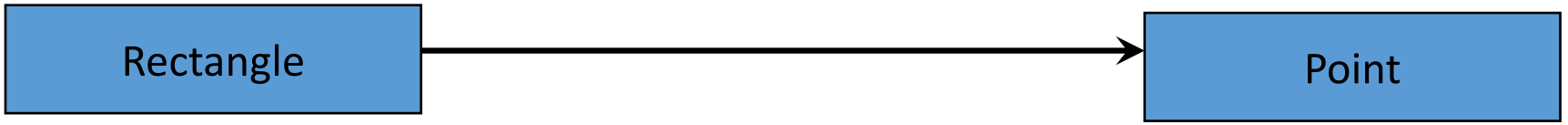
- `Rectangle` class passes responsibility for determining equality and moving to `Point` class invoking its `equals` and `move` methods

Representation of Composition in UML

- *Represented as a solid line connecting both classes*
- ***Bi-directional*** (default)
  - Both classes are aware of each other and their relationship

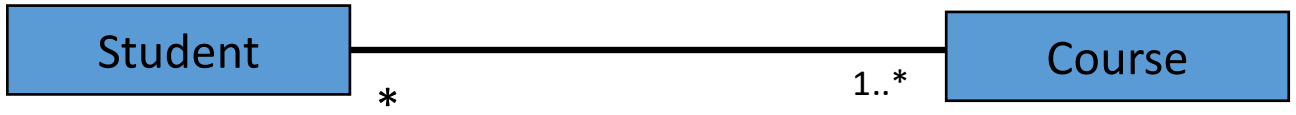


  - **Not the case here**
- Uni-directional
  - Both classes are related, but only one class knows that the relationship exists

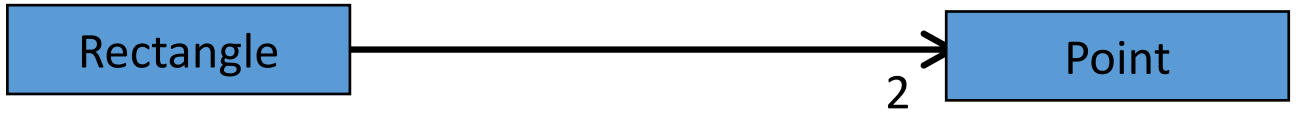


Association Relationships - Multiplicity

- We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association
  - E.g., a *Student* has one or more *Course*:
  - And a Course has zero or more Students



- To show that every *Rectangle* has two *Point*:



- Potential Multiplicity Values
  - 0..1 – zero or one
  - 1 (default) – exactly one
  - 0..* - zero or more (or just *)
  - n – exactly n
  - n..p – between ***n*** and ***p***

Composition Considerations

- Composition is a fundamental way to reuse code, but there are coding considerations when composition is used
- With composition, *Rectangle* becomes a client of the *Point* class.
- *Rectangle* class has no special privileges with respect to *Point* class
- *Rectangle* class should delegate responsibility to the Point class whenever possible
  - Be lazy!
  - Do not try to do the work that belongs to another object/class

Using and Misusing References

- When writing a program, it is very important to insure that private instance variables remain truly private
- For a primitive type instance variable, just adding the `private` modifier to its declaration should insure that there will be no ***privacy leaks***
  - In the sense that the private state of an instance cannot be changed in other classes
  - Change must be made through method invocation
- For a class type instance variable, adding the `private` modifier alone is not sufficient.

Pitfall: Privacy Leaks

- **Privacy leaks** typically occur with incorrectly defined mutator or accessor methods

```
public Point getOrigin() {
  return _origin;
}
```

Dangerous

- Why?
- Now, we can directly change internal state of a rectangle instance

```
Rectangle r;
// …
Point p = r.getOrigin();
p.setX(5);  // Privacy leak!
```

Solution?

```
public Point getOrigin() {
  return new Point(_origin.getX(),
                   _origin.getY());
}
```

Safe

Another Solution: Immutable Objects

- It is possible to have an accessor method that returns the reference to the field without incurring in a privacy leak

- When is this valid?

```
public Type getSomeref() {
  return _someRef;
}
```

- When *Type* is *immutable*
  - Each instance is a constant.
  - Its state cannot be changed after it is created
    - *String* is immutable
    - All Wraper classes are immutable

Immutable Point class

- Original methods of Point
  - getX, getY, setX, setY, move, equals
- Immutable Point version
  - Cannot offer
    - setX(), setY()
  - move()?
    - Must change it

```
public class Point{
  private int _x;
  private int _y;

  public Point(int x, int y) {
    _x = x;
    _y = y;
  }

  public int getX() {
    return _x;
  }

  public int getY() {
    return _y;
  }

  public Point move(int dx, int dy) {
    return new Point(_x + dx,_y + dy);
  }

  public boolean equals(Point r) {
    // remains the same
  }
}
```

Composition with Arrays

- Just as a class type can be used as an instance variable, arrays can also be used as instance variables
- We can define an array with a primitive base type

```
private double[] _grades;
```

- Or, an array with a class base type.

```
private Point[] _points;
```

Privacy Leaks with Array Instance Variables

- If an accessor method is provided for the array, special care must be taken just as when an accessor returns a reference to any private object.

```
public Point[] getPoints() {
  return _points;
}
```

- The example above can result in a privacy leak
- Why?

```
Type t;
// …
Point[] p = t.getPoints();
p[0] = new Point(1, 1);    // Privacy leak? YES
p = new Point[5];          // Privacy leak? NO
```

Privacy Leaks with Array Instance Variables - Solution

- Problem: accessor method simply returns a reference to the private array field
- Solution?
- If array contains primitive values or immutable objects,
  - accessor method returns a reference to a ***copy*** of the private array object

```
public Point[] getPoint() {
  Point[] temp = new Point[_points.length];

  for (int i = 0; i < _point.length; i++) {
    temp[i] = _points[i];
  }

  return temp;
}  // using the immutable Point Version
```

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a mutable class as its base type,
- Then do a **deep copy**
  - copies must be made of each object in the array when the array is copied.

```
public Point[] getPoints() {
  Point[] temp = new Point[_points.length];
  for (int i = 0; i < _points.length; i++) {
    Point p = _points[i];
    temp[i] = new Point(p.getX(), p.getY());
  }

  return temp;
} //using the mutable Point Version
```

But what if…

- …the user really wants to change the array within the class?
  - The user shouldn’t know that the class uses an array
  - The array must represent some abstract data element in the class
  - Provide a method that changes the abstract data element without revealing the existence of an array

Remember…