Composition

Composition

- recall that an object of type **`X`** that is composed of an object of type **`Y`** means
  - **`X`** has-a **`Y`** object *and*
  - **`X`** owns the **`Y`** object
- in other words

> the **`X`** object has exclusive access to its **`Y`** object

Composition

the **`X`** object has exclusive access to its **`Y`** object

- this means that the **`X`** object will generally not share references to its **`Y`** object with clients
  - constructors will create new **`Y`** objects
  - accessors will return references to new **`Y`** objects
  - mutators will store references to new **`Y`** objects
- the "new **`Y`** objects" are called *defensive copies*

Composition & the Default Constructor

the **`X`** object has exclusive access to its **`Y`** object

- if a default constructor is defined it must create a suitable **`Y`** object

```
public X()
{
  // create a suitable Y; for example
  this.y = new Y( /* suitable arguments */ );
}
```

defensive copy

Test Your Knowledge

1. Re-implement Triangle so that it is a composition of 3 points. Start by adding a default constructor to **`Triangle`** that creates 3 new **`Point`** objects with suitable values.

Composition & Copy Constructor

the **X** object has exclusive access to its **Y** object

- if a copy constructor is defined it must create a new **Y** that is a deep copy of the other **X** object's **Y** object

```
public X(X other)
{
  // create a new Y that is a copy of other.y
  this.y = new Y(other.getY());
}
```

defensive copy

Composition & Copy Constructor

- what happens if the **`X`** copy constructor does not make a deep copy of the other **`X`** object's **`Y`** object?

```
// don't do this
public X(X other)
{
  this.y = other.y;
}
```

  - every **`X`** object created with the copy constructor ends up sharing its **`Y`** object
    - if one **`X`** modifies its **`Y`** object, all **`X`** objects will end up with a modified **`Y`** object
    - this is called a privacy leak

Test Your Knowledge

1. Suppose `Y` is an immutable type. Does the `X` copy constructor need to create a new `Y`? Why or why not?

2. Implement the `Triangle` copy constructor.

3. Suppose you have a **`Triangle`** copy constructor and **`main`** method like so:

```
public Triangle(Triangle t)
{  this.pA = t.pA;  this.pB = t.pB;  this.pC = t.pC; }

public static void main(String[] args) {
  Triangle t1 = new Triangle();
  Triangle t2 = new Triangle(t1);
  t1.getA().set( -100.0, -100.0, 5.0 );
  System.out.println( t2.getA() );
}
```

What does the program print? How many **`Point`** objects are there in memory? How many **`Point`** objects should be in memory?

Composition & Other Constructors

the **X** object has exclusive access to its **Y** object

- a constructor that has a **Y** parameter must first deep copy and then validate the **Y** object

```
public X(Y y)
{
  // create a copy of y
  Y copyY = new Y(y);
  // validate; will throw an exception if copyY is invalid
  this.checkY(copyY);
  this.y = copyY;
}
```

defensive copy

Composition and Other Constructors

- why is the deep copy required?

> the **`X`** object has exclusive access to its **`Y`** object

  - if the constructor does this

```
// don't do this for composition
public X(Y y) {
   this.y = y;
}
```

    then the client and the **`X`** object will share the same **`Y`** object
    - this is called a privacy leak

Test Your Knowledge

1. Suppose **`Y`** is an immutable type. Does the **`X`** constructor need to copy the other **`X`** object's **`Y`** object? Why or why not?

2. Implement the following **`Triangle`** constructor:

```
/**
 * Create a Triangle from 3 points
 * @param p1 The first point.
 * @param p2 The second point.
 * @param p3 The third point.
 * @throws IllegalArgumentException if the 3 points are
 *         not unique
 */
```

Triangle has a class invariant: the 3 points of a Triangle are unique

Composition and Accessors

the **`X`** object has exclusive access to its **`Y`** object

- never return a reference to a field; always return a deep copy

```
public Y getY()
{
  return new Y(this.y);
}
```

defensive copy

Composition and Accessors

- why is the deep copy required?

> the **`X`** object has exclusive access to its **`Y`** object

  - if the accessor does this

```
// don't do this for composition
public Y getY() {
  return this.y;
}
```

    then the client and the **`X`** object will share the same **`Y`** object
    - this is called a privacy leak

Test Your Knowledge

1. Suppose **`Y`** is an immutable type. Does the **`X`** accessor need to copy it's **`Y`** object before returning it? Why or why not?

2. Implement the following 3 **`Triangle`** accessors:

```
/**
 * Get the first/second/third point of the triangle.
 * @return The first/second/third point of the triangle
 */
```

Test Your Knowledge

3. Given your **`Triangle`** accessors from question 2, can you write an improved **`Triangle`** copy constructor that does not make copies of the point attributes?

Composition and Mutators

the **X** object has exclusive access to its **Y** object

- if **X** has a method that sets its **Y** object to a client-provided **Y** object then the method must make a deep copy of the client-provided **Y** object and validate it

```
public void setY(Y y)
{
  Y copyY = new Y(y);
  // validate; will throw an exception if copyY is invalid
  this.checkY(copyY);
  this.y = copyY;
}
```

defensive copy

Composition and Mutators

- why is the deep copy required?

> the **`X`** object has exclusive access to its **`Y`** object

  - if the mutator does this

```
// don't do this for composition
public void setY(Y y) {
   this.y = y;
}
```

    then the client and the **`X`** object will share the same **`Y`** object
    - this is called a privacy leak

Test Your Knowledge

1. Suppose **`Y`** is an immutable type. Does the **`X`** mutator need to copy the **`Y`** object? Why or why not? Does it need to the validate the **`Y`** object?

2. Implement the following 3 **`Triangle`** mutators:

```
/**
 * Set the first/second/third point of the triangle.
 * @param p The desired first/second/third point of
 *          the triangle.
 * @return true if the point could be set;
 *         false otherwise
 */
```

Triangle has a class invariant: the 3 points of a Triangle are unique

Price of Defensive Copying

- defensive copies are often required, but the price of defensive copying is time and memory needed to create and garbage collect defensive copies of objects

- recall the triangle demo from the previous lecture
  - a triangle was an aggregation of three points
    - because it was an aggregation, the client could change the location of a point without asking the triangle

Price of Defensive Copying

- if triangle is composed of three points, there is no way for the client to directly change the location of a point
- to change the location of a point, the client must either:

A.

i. ask the triangle for the point (1 defensive copy)
ii. change the location of the point
iii. ask the triangle to change its point (1 defensive copy)

B.

i. keep an independent copy of the point
ii. change the location of the copy
iii. ask the triangle to change its point (1 defensive copy)

```
pointB = new Point(0.0, 1.0, -3.0);
tri = new Triangle(new Point(-1.0, -1.0, -3.0),
                   pointB,
                   new Point(2.0, 0.0, -3.0));
```

triangle makes defensive copies of all three points

```
// Draw triangle
gl.glBegin(GL2.GL_TRIANGLES);
gl.glColor3f(0.0f, 1.0f, 1.0f); // set the color
gl.glVertex3d(tri.getA().getX(),
              tri.getA().getY(),
              tri.getA().getZ());
gl.glVertex3d(tri.getB().getX(),
              tri.getB().getY(),
              tri.getB().getZ());
gl.glVertex3d(tri.getC().getX(),
              tri.getC().getY(),
              tri.getC().getZ());
gl.glEnd();
```

draw the triangle by asking **`tri`** for the coordinates of each of its points

```
// the client moves its point, then asks the triangle to change
delta += 0.05f;
pointB.setY(1.0 + Math.sin(delta));
tri.setB(pointB);
```

triangle makes defensive copy

Price of Defensive Copying

- run triangle demo using composition here
  - that's a lot of points being created!