A Singleton Puzzle: What is Printed?

```
public class Elvis {
  public static final Elvis INSTANCE = new Elvis();
  private final int beltSize;
  private static final int CURRENT_YEAR =
    Calendar.getInstance().get(Calendar.YEAR);

  private Elvis() { this.beltSize = CURRENT_YEAR – 1930; }

  public int getBeltSize() { return this.beltSize; }

  public static void main(String[] args) {
    System.out.println("Elvis has a belt size of " +
                       INSTANCE.getBeltSize());
  }
}
```

from Java Puzzlers by Joshua Bloch and Neal Gafter

A Singleton Puzzle: Solution

- **`Elvis has a belt size of -1930`** is printed
- to solve the puzzle you need to know how Java initializes classes (JLS 12.4)
- the call to **`main()`** triggers initialization of the **`Elvis`** class (because **`main()`** belongs to the class Elvis)
- the static attributes **`INSTANCE`** and **`CURRENT_YEAR`** are first given default values (**`null`** and **`0`**, respectively)
- then the attributes are initialized in order of appearance

1. `public static final Elvis INSTANCE = new Elvis();`
2. `this.beltSize = CURRENT_YEAR – 1930;`

   `CURRENT_YEAR == 0`

   at this point
3. `private static final int CURRENT_YEAR =`
   `Calendar.getInstance().get(Calendar.YEAR);`

- the problem occurs because initializing **`INSTANCE`** requires a valid **`CURRENT_YEAR`**
- solution: move **`CURRENT_YEAR`** before **`INSTANCE`**

Aggregation and Composition

[notes Chapter 4]

Aggregation and Composition

- the terms aggregation and composition are used to describe a relationship between objects

- both terms describe the *has-a* relationship
  - the university has-a collection of departments
  - each department has-a collection of professors

Aggregation and Composition

- composition implies ownership
  - if the university disappears then all of its departments disappear
  - a university is a *composition* of departments
- aggregation does not imply ownership
  - if a department disappears then the professors do not disappear
  - a department is an *aggregation* of professors

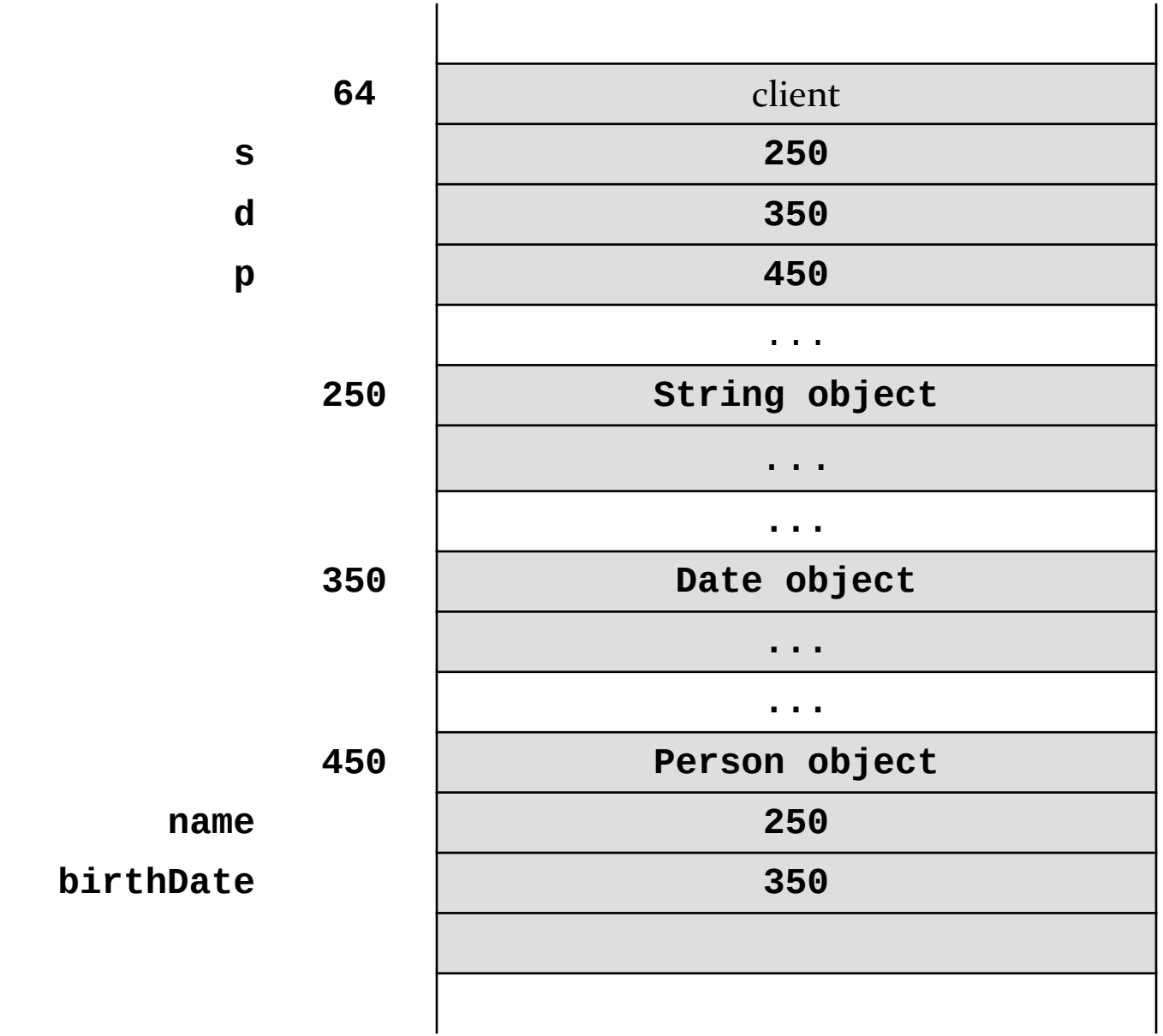
Aggregation

- suppose a **`Person`** has a name and a date of birth

```
public class Person {
  private String name;
  private Date birthDate;

  public Person(String name, Date birthDate) {
    this.name = name;
    this.birthDate = birthDate;
  }

  public Date getBirthDate() {
    return birthDate;
  }
}
```

- the **`Person`** example uses aggregation
  - notice that the constructor does not make a copy of the name and birth date objects passed to it
  - the name and birth date objects are shared with the client
  - both the client and the **`Person`** instance are holding references to the same name and birth date

```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(91, 2, 26);  // March 26, 1991
Person p = new Person(s, d);
```


64
client
s
250
d
350
p
450
...
250
String object
...
...
350
Date object
...
...
450
Person object
name
250
birthDate
350

- what happens when the client modifies the **`Date`** instance?

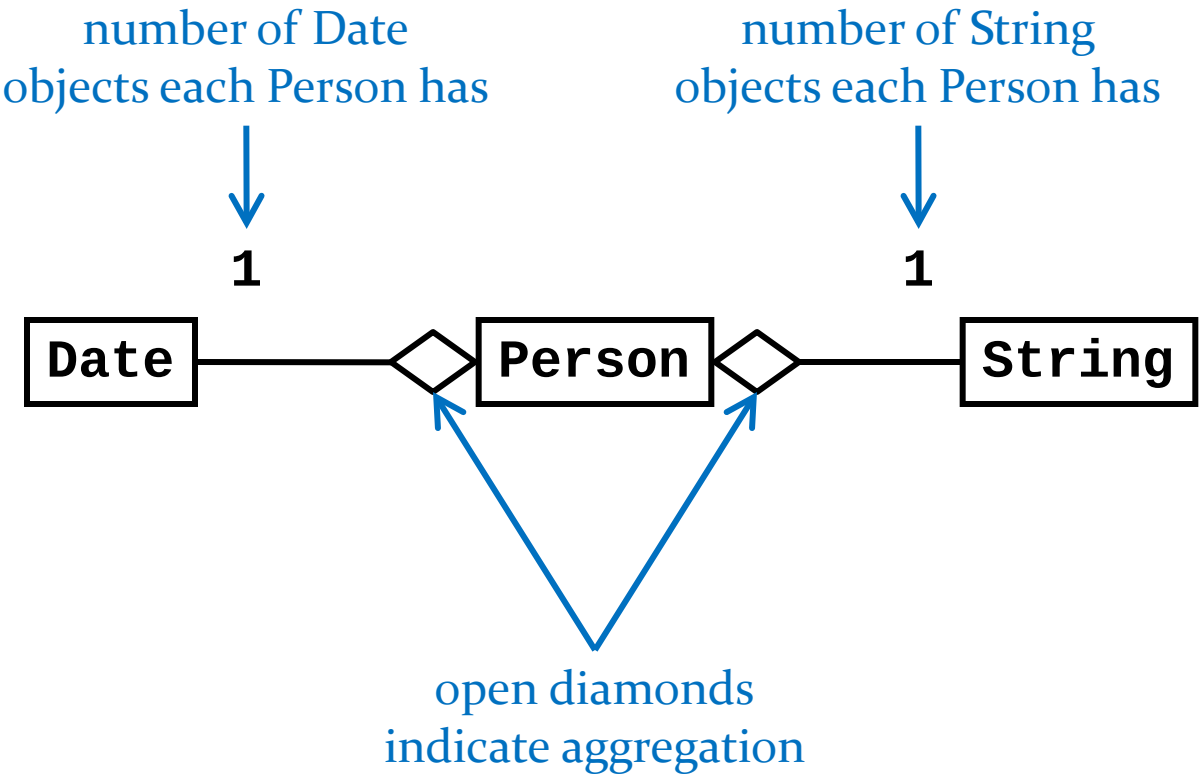
```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(90, 2, 26);  // March 26, 1990
Person p = new Person(s, d);

d.setYear(95);                 // November 3, 1995
d.setMonth(10);
d.setDate(3);
System.out.println( p.getBirthDate() );
```

  - prints **`Fri Nov 03 00:00:00 EST 1995`**

- because the **`Date`** instance is shared by the client and the **`Person`** instance:
  - the client can modify the date using **`d`** and the **`Person`** instance **`p`** sees a modified **`birthDate`**
  - the **`Person`** instance **`p`** can modify the date using **`birthDate`** and the client sees a modified date **`d`**

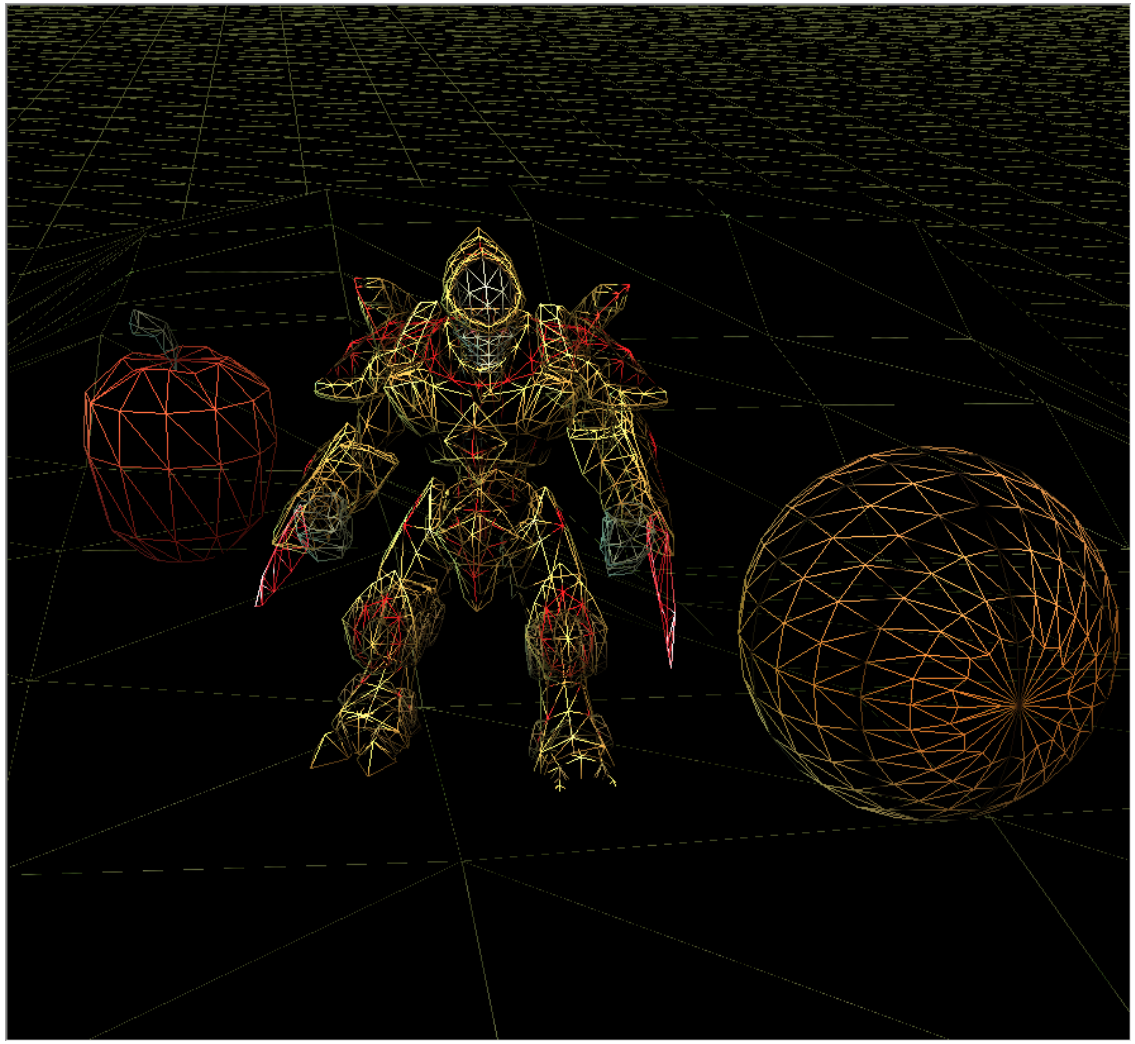
- note that even though the **`String`** instance is shared by the client and the **`Person`** instance **`p`**, neither the client nor **`p`** can modify the **`String`**
  - immutable objects make great building blocks for other objects
  - they can be shared freely without worrying about their state

UML Class Diagram for Aggregation



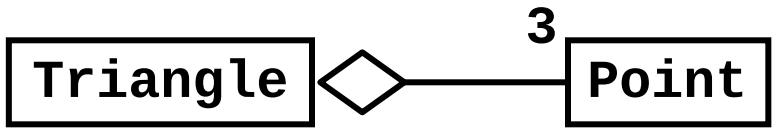
Another Aggregation Example

- 3D videogames use models that are a three-dimensional representations of geometric data
  - the models may be represented by:
    - three-dimensional points (particle systems)
    - simple polygons (triangles, quadrilaterals)
    - smooth, continuous surfaces (splines, parametric surfaces)
    - an algorithm (procedural models)
- rendering the objects to the screen usually results in drawing triangles
  - graphics cards have specialized hardware that does this very fast



Aggregation Example

- a **`Triangle`** has 3 three-dimensional **`Point`**s



Triangle
+ Triangle(Point, Point, Point)
+ getA() : Point + getB() : Point + getC() : Point + setA(Point) : void + setB(Point) : void + setC(Point) : void

Point
+ Point(double, double, double)
+ getX() : double + getY() : double + getZ() : double + setX(double) : void + setY(double) : void + setZ(double) : void

Triangle

```
// attributes and constructor

public class Triangle {

  private Point pA;
  private Point pB;
  private Point pC;

  public Triangle(Point c, Point b, Point c) {
    this.pA = a;
    this.pB = b;
    this.pC = c;
  }
```

Triangle

```
// accessors

public Point getA() {
  return this.pA;
}

public Point getB() {
  return this.pB;
}

public Point getC() {
  return this.pC;
}
```

Triangle

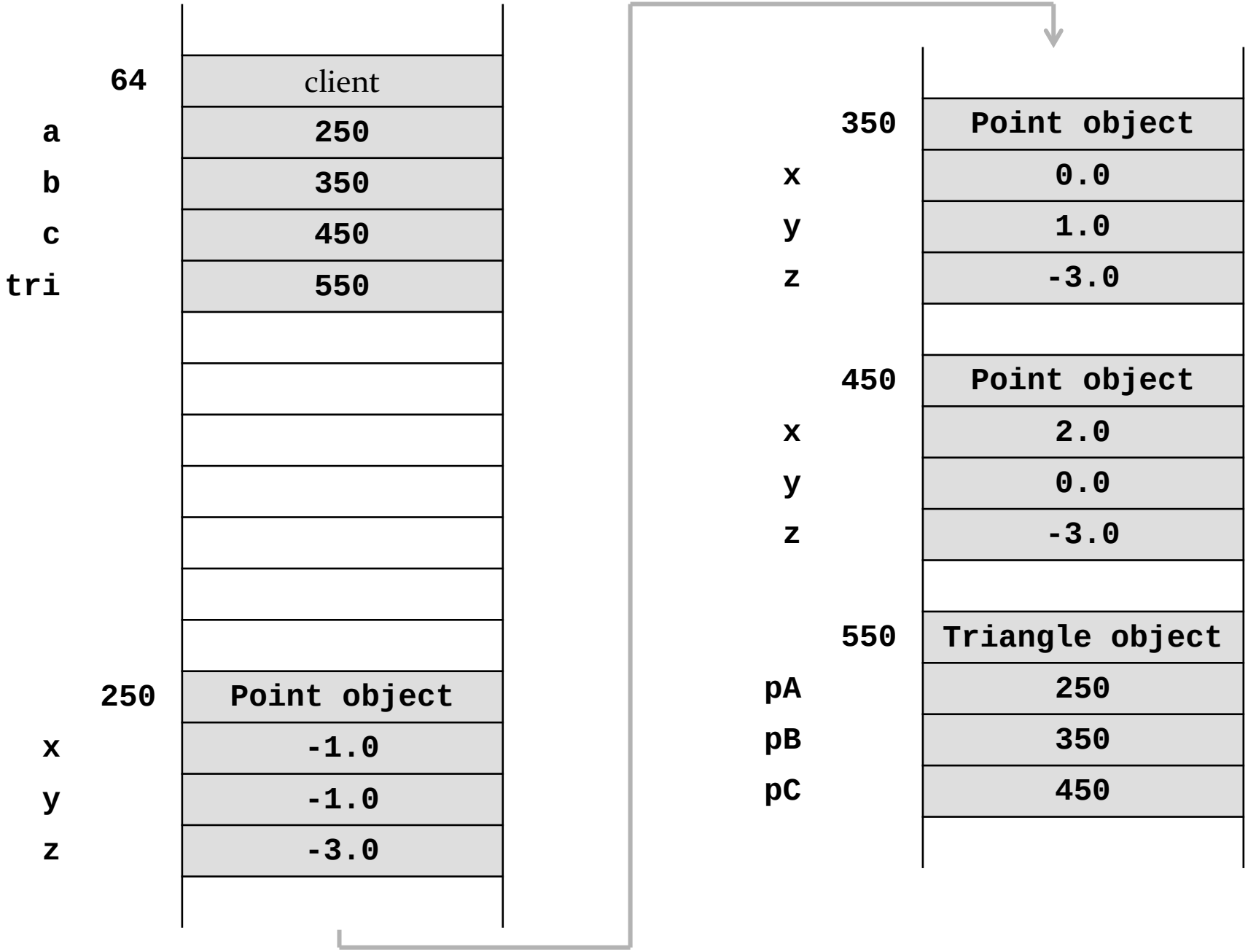
```
  // mutators

  public void setA(Point p) {
    this.pA = p;
  }

  public void setB(Point p) {
    this.pB = p;
  }

  public void setC(Point p) {
    this.pC = p;
  }
}
```

Triangle Aggregation

- implementing **`Triangle`** is very easy
- attributes (3 **`Point`** references)
  - are references to existing objects provided by the client
- accessors
  - give clients a reference to the aggregated **`Point`**s
- mutators
  - set attributes to existing **`Point`**s provided by the client

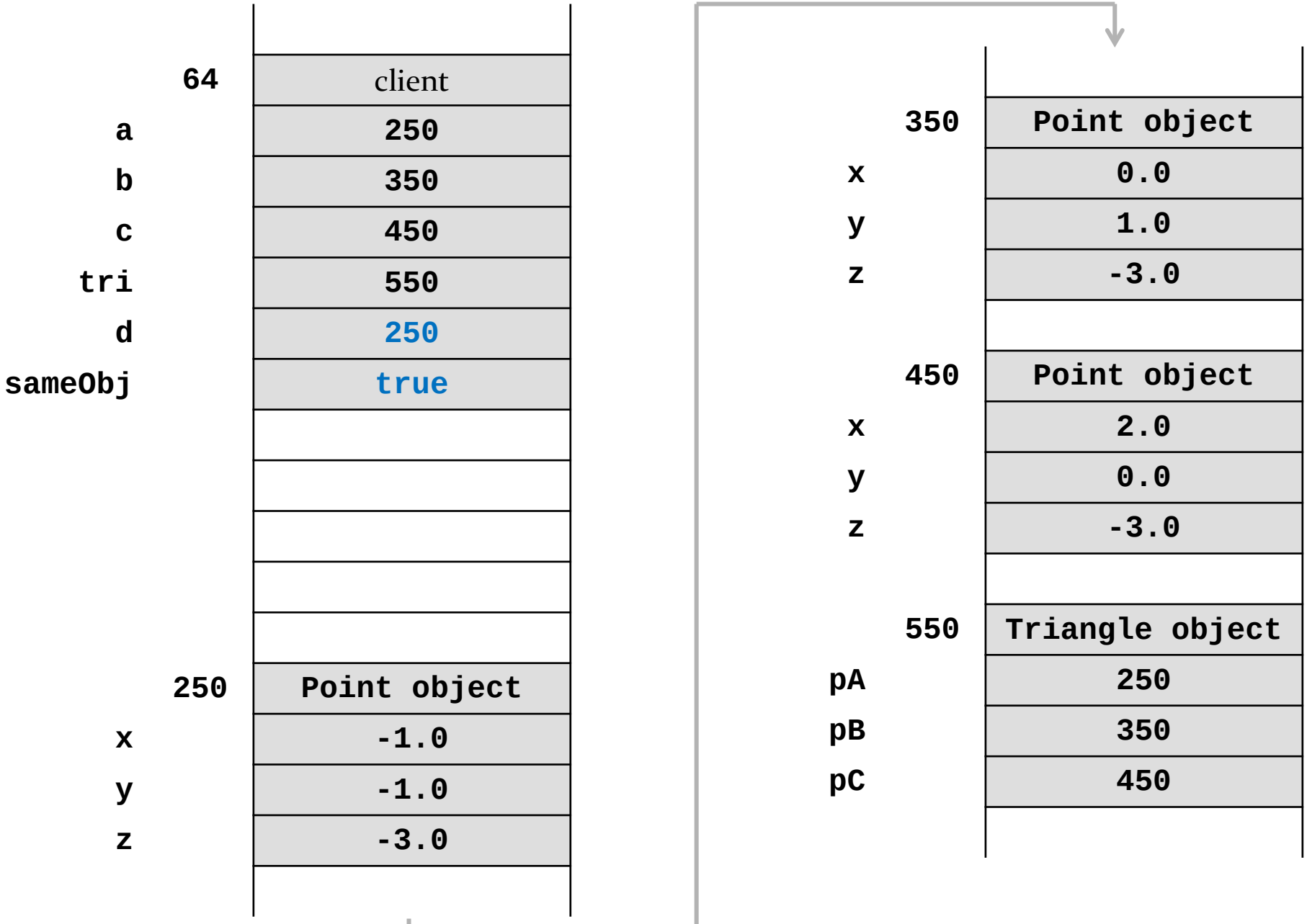
- we say that the **`Triangle`** attributes are *aliases*

```
// client code

Point a = new Point(-1.0, -1.0, -3.0);
Point b = new Point(0.0, 1.0, -3.0);
Point c = new Point(2.0, 0.0, -3.0);
Triangle tri = new Triangle(a, b, c);
```


64
client
a
250
b
350
c
450
tri
550
250
Point object
x
-1.0
y
-1.0
z
-3.0
350
Point object
x
0.0
y
1.0
z
-3.0
450
Point object
x
2.0
y
0.0
z
-3.0
550
Triangle object
pA
250
pB
350
pC
450

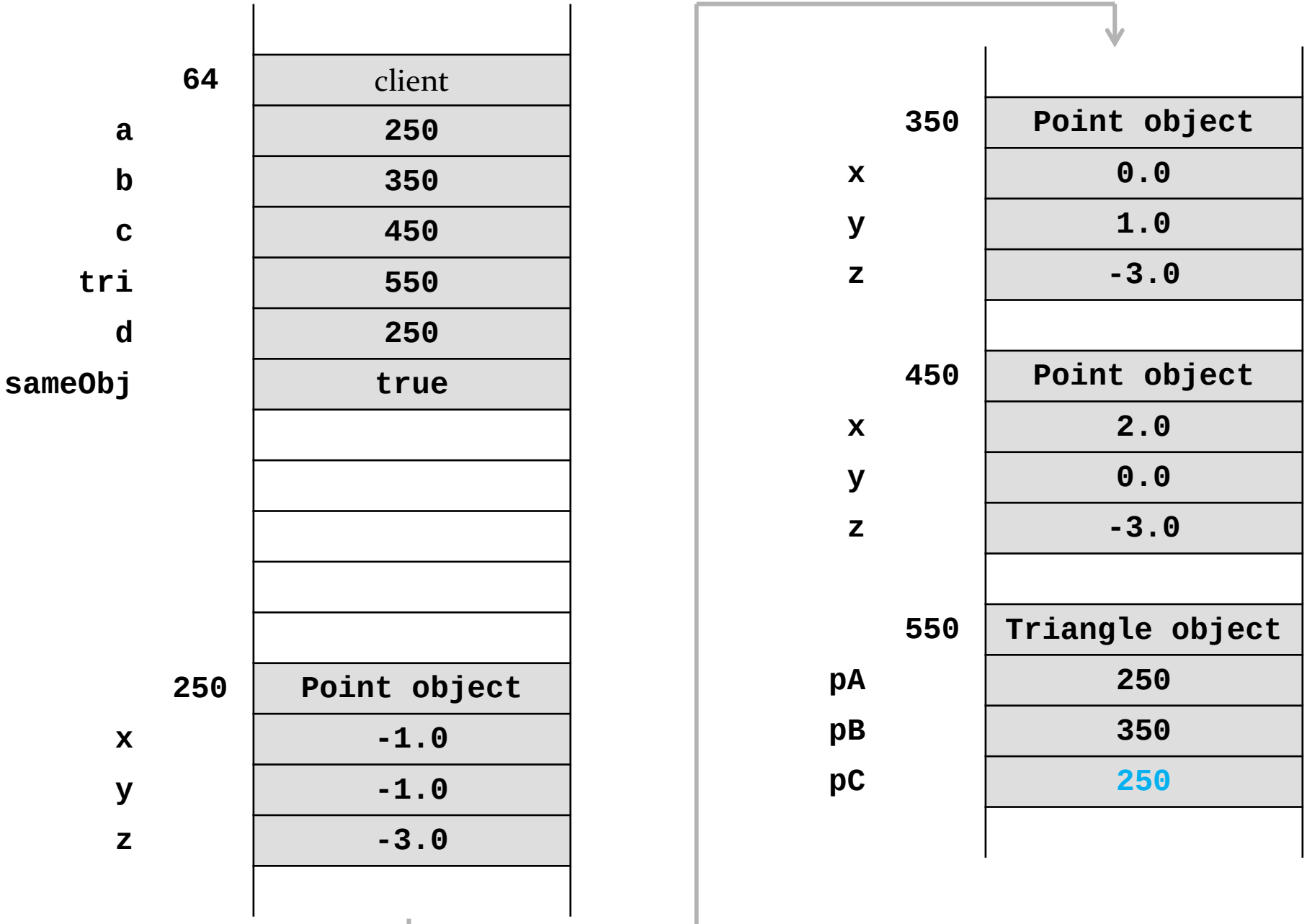
```
// client code

Point a = new Point(-1.0, -1.0, -3.0);
Point b = new Point(0.0, 1.0, -3.0);
Point c = new Point(2.0, 0.0, -3.0);
Triangle tri = new Triangle(a, b, c);
Point d = tri.getA();
boolean sameObj = a == d;
```

client asks the triangle for one of the triangle points and checks if the point is the same object that was used to create the triangle


64
client
a
250
b
350
c
450
tri
550
d
250
sameObj
true
250
Point object
x
-1.0
y
-1.0
z
-3.0
350
Point object
x
0.0
y
1.0
z
-3.0
450
Point object
x
2.0
y
0.0
z
-3.0
550
Triangle object
pA
250
pB
350
pC
450

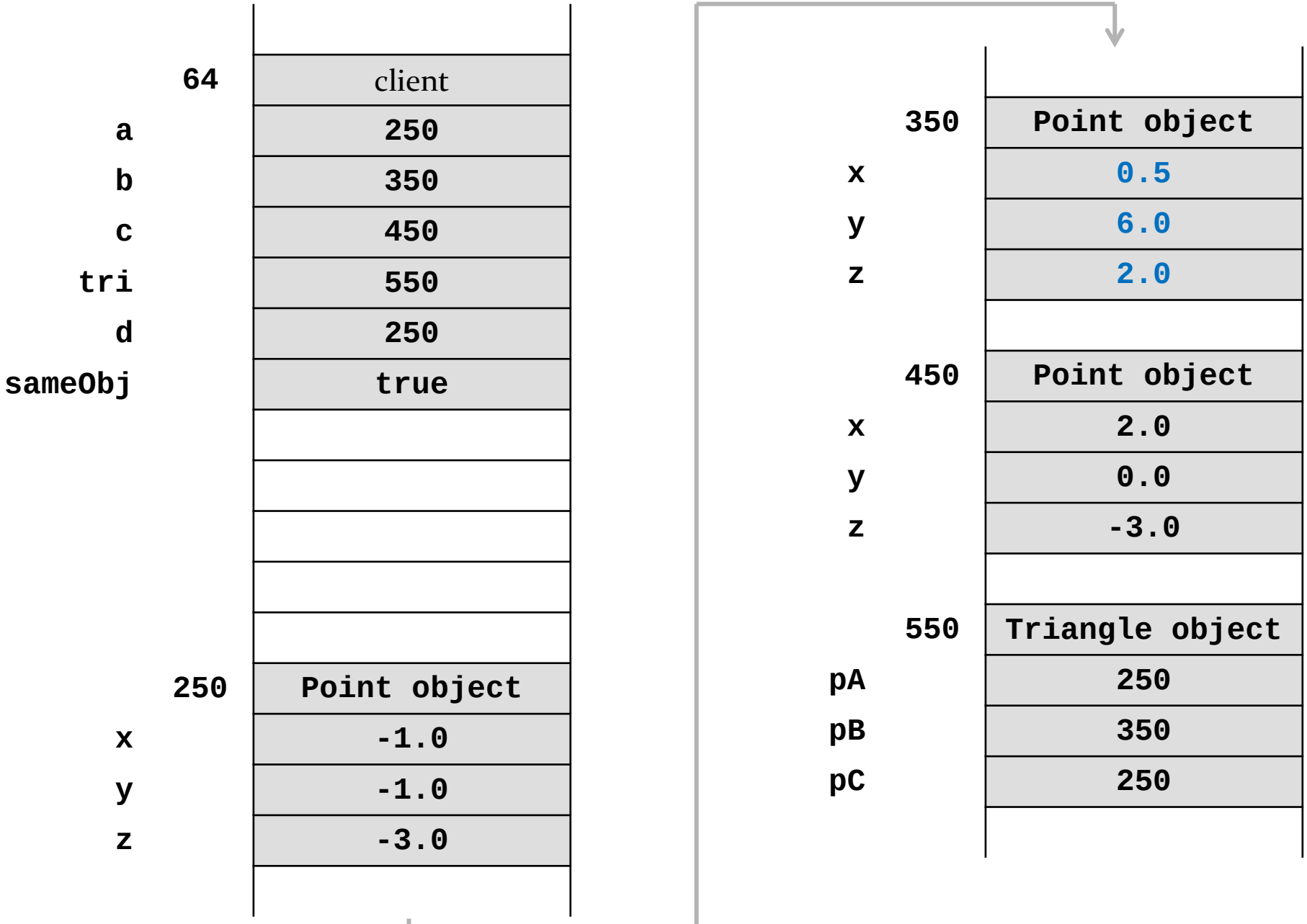
```
// client code

Point a = new Point(-1.0, -1.0, -3.0);
Point b = new Point(0.0, 1.0, -3.0);
Point c = new Point(2.0, 0.0, -3.0);
Triangle tri = new Triangle(a, b, c);
Point d = tri.getA();
boolean sameObj = a == d;
tri.setC(d);
```

client asks the triangle to set one point of the triangle to **d**


64
client
a
250
b
350
c
450
tri
550
d
250
sameObj
true
250
Point object
x
-1.0
y
-1.0
z
-3.0
350
Point object
x
0.0
y
1.0
z
-3.0
450
Point object
x
2.0
y
0.0
z
-3.0
550
Triangle object
pA
250
pB
350
pC
250

```
// client code

Point a = new Point(-1.0, -1.0, -3.0);
Point b = new Point(0.0, 1.0, -3.0);
Point c = new Point(2.0, 0.0, -3.0);
Triangle tri = new Triangle(a, b, c);
Point d = tri.getA();
boolean sameObj = a == d;
tri.setC(d);
b.setX(0.5);
b.setY(6.0);
b.setZ(2.0);
```

client changes the coordinates of one of the points (without asking the triangle for the point first)


64
client
a
250
b
350
c
450
tri
550
d
250
sameObj
true
250
Point object
x
-1.0
y
-1.0
z
-3.0
350
Point object
x
0.5
y
6.0
z
2.0
450
Point object
x
2.0
y
0.0
z
-3.0
550
Triangle object
pA
250
pB
350
pC
250

Triangle Aggregation

- if a client gets a reference to one of the triangle's points, then the client can change the position of the point *without asking the triangle*

- run demo program in class here

```
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));

// 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();

// the client moves a point without help from the triangle
delta += 0.05f;
pointB.setY(1.0 + Math.sin(delta));
```

client and triangle share a reference to **pointB**

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

client uses **pointB** to change the point coordinates

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, and only the **X** object, is responsible for its **Y** object

Composition

the **X** object, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for its **Y** object

- never return a reference to an attribute; 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, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for 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, and only the **X** object, is responsible for 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