Aggregation and Composition

Based on slides by Prof. Burton Ma

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

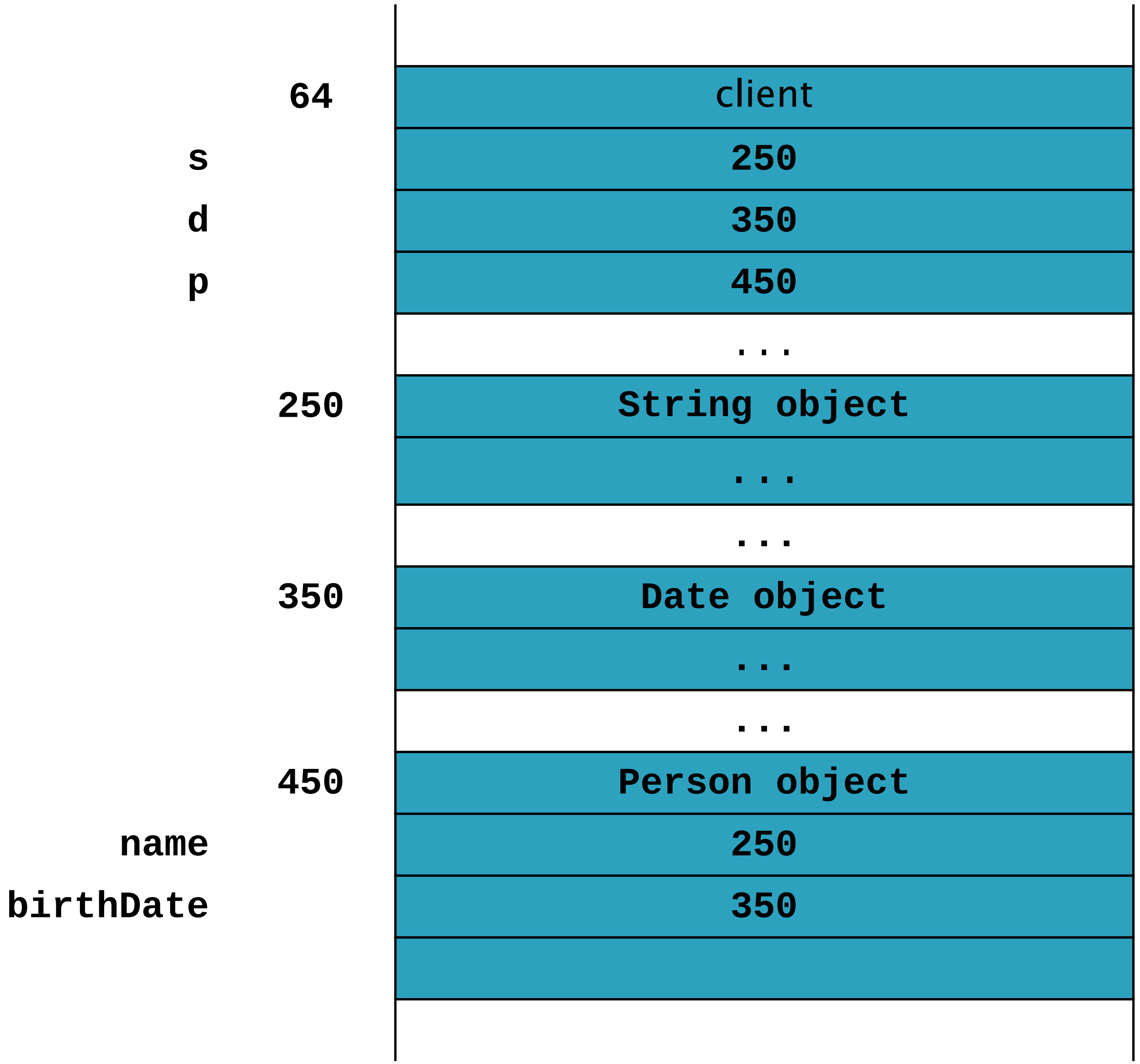
Aggregation

- 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?

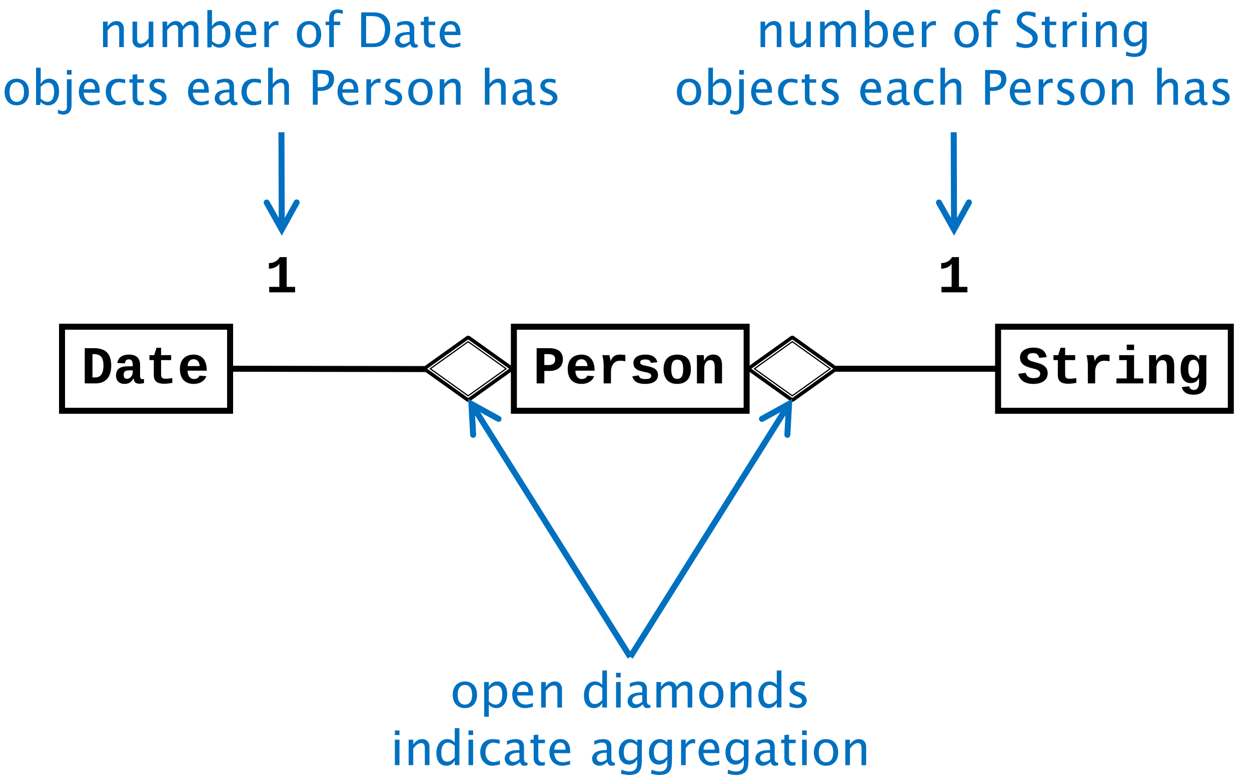
```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(90, 2, 26);  // March 26, 1990
Person p = new Person(s, d);

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

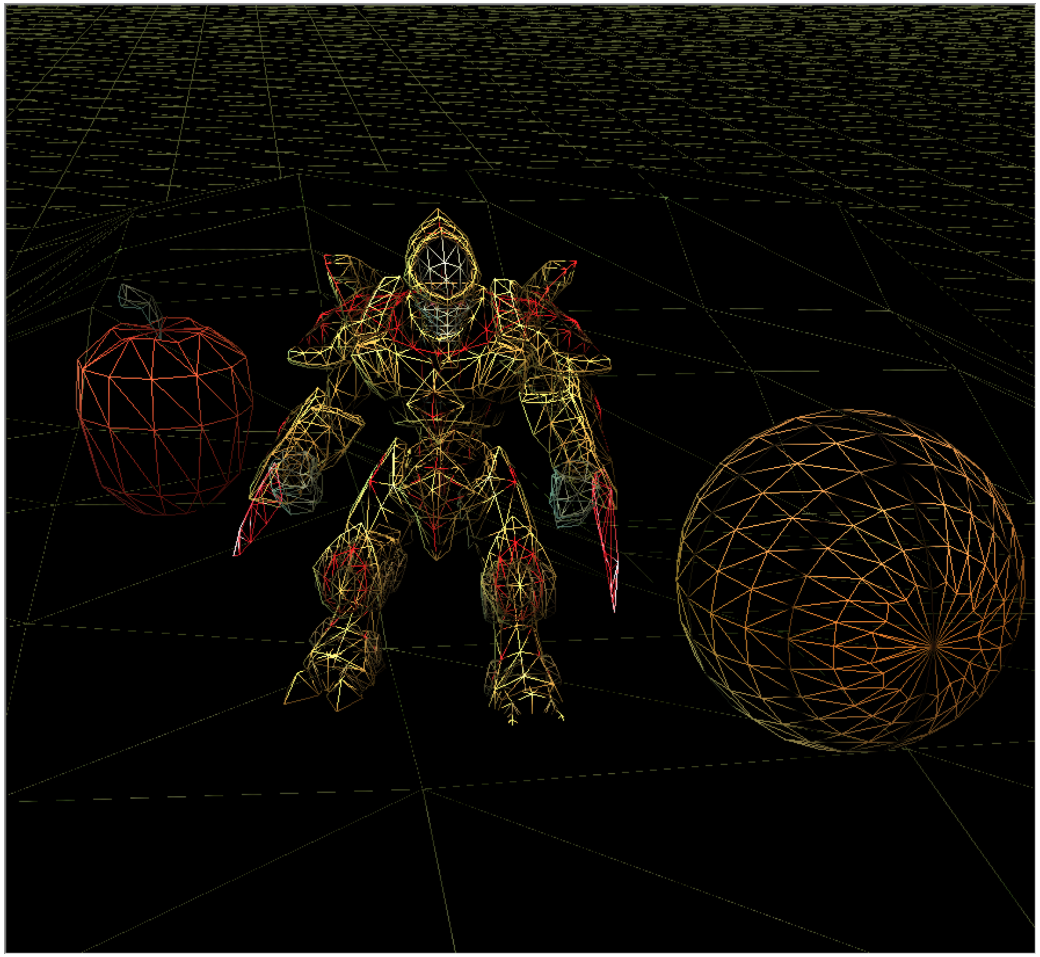

number of Date objects each Person has

number of String objects each Person has

open diamonds indicate aggregation

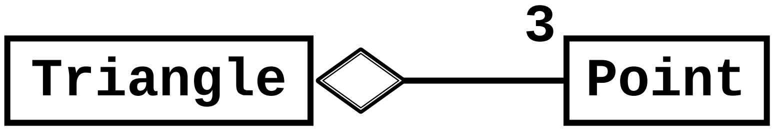
Another Aggregation Example

- 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 **Points**



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 a, 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;
}
```

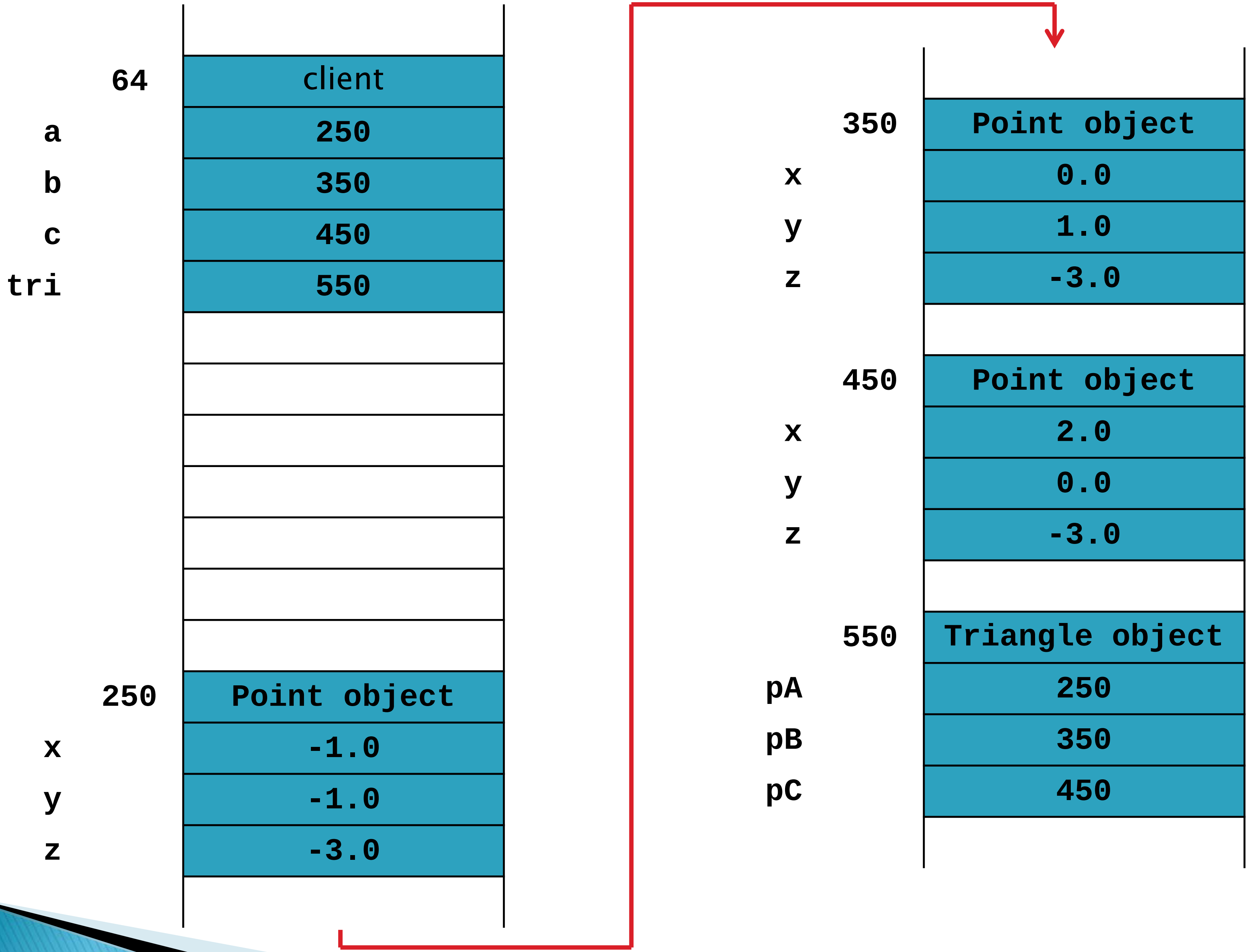
Triangle

```
  // 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);
```

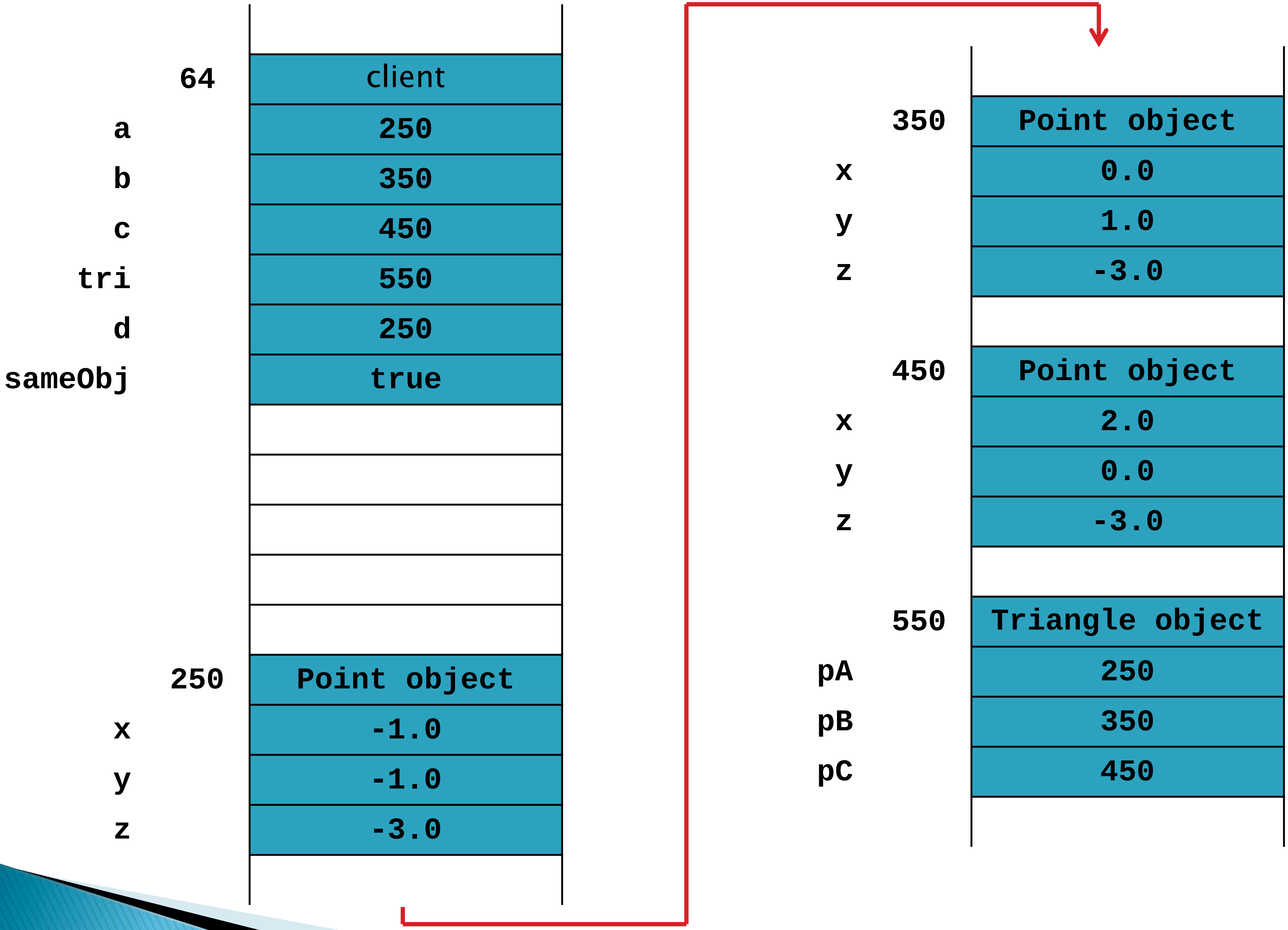


	64	client
a		250
b		350
c		450
tri		550
	250	Point object
x		-1.0
y		-1.0
z		-3.0

	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

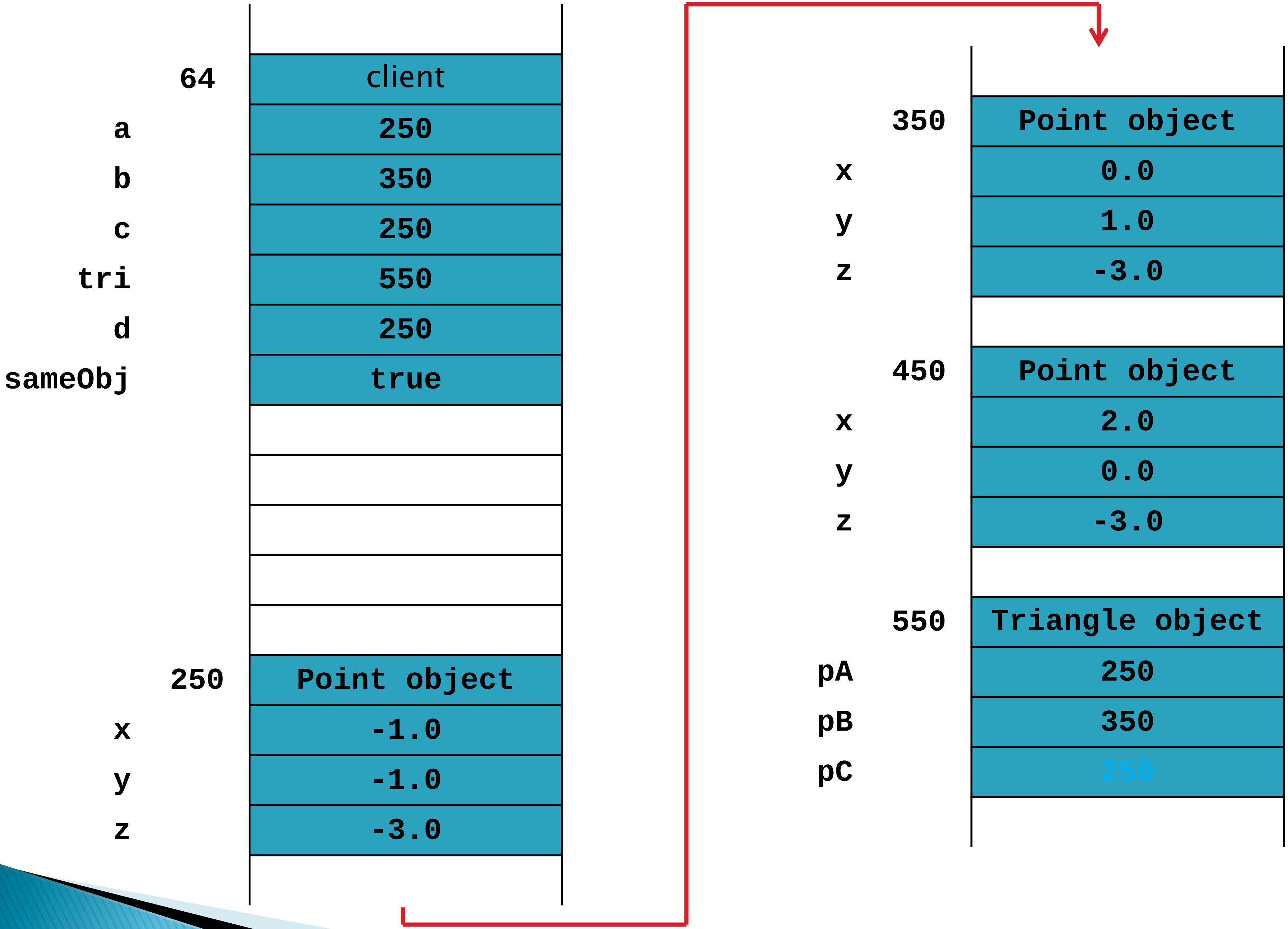
pA 250

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**

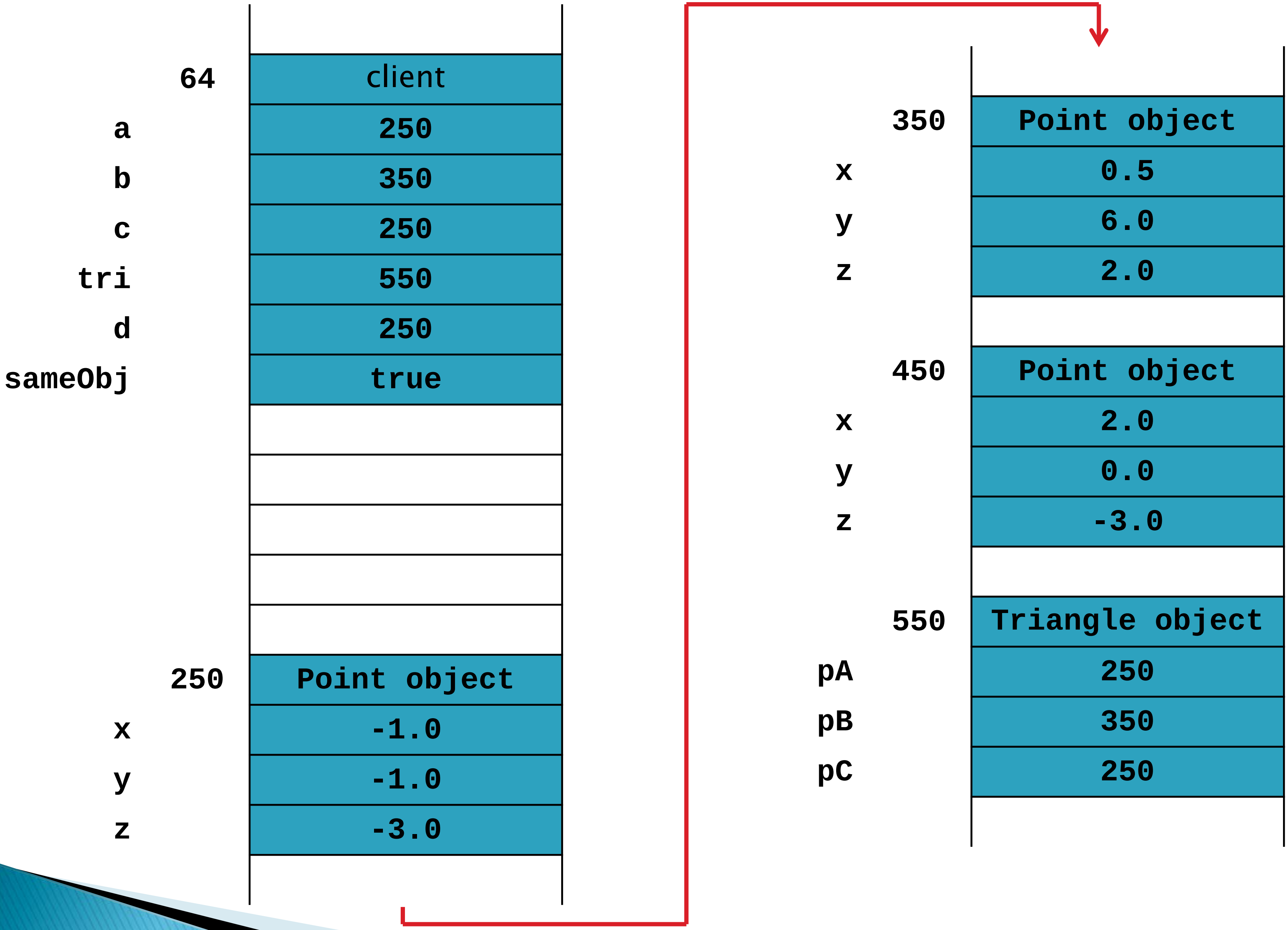


	64	client
a		250
b		350
c		250
tri		550
d		250
sameObj		true
	250	Point object
x		-1.0
y		-1.0
z		-3.0

	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		250
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*

```
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));
```

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

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

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

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

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
    - What is this an example of?

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 a privacy leak

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 a privacy leak

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 a privacy leak

Period Class

- Adapted from Effective Java by Joshua Bloch
  - Available online at http://www.informit.com/articles/article.aspx?p=31551&seqNum=2

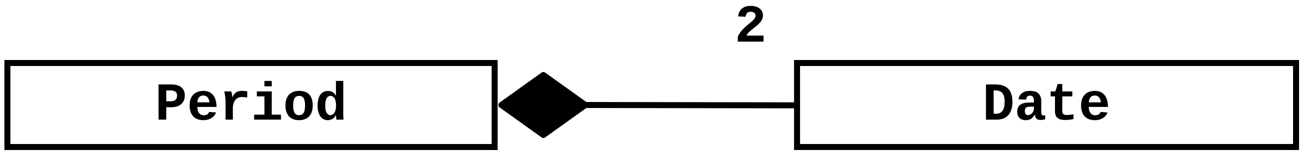
- We want to implement a class that represents a period of time
  - A period has a start time and an end time
    - End time is always after the start time

Period Class

- We want to implement a class that represents a period of time
  - Has-a: **`Date`** representing the start of the time period
  - Has-a: **`Date`** representing the end of the time period
  - Class invariant: start of time period is always prior to the end of the time period

- Class invariant
  - Some property of the state of the object that is established by a constructor and maintained between calls to public methods

Period Class



Period is a compostion of two **Date** objects

```
public final class Period
{
  private Date start;
  private Date end;

  /**
   * @param start beginning of the period.
   * @param end end of the period; must not precede start.
   * @throws IllegalArgumentException if start is after end.
   * @throws NullPointerException if start or end is null
   */
  public Period(Date start, Date end) {
    if (start.compareTo(end) > 0) {
      throw new IllegalArgumentException("start after end");
    }
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
  }
```

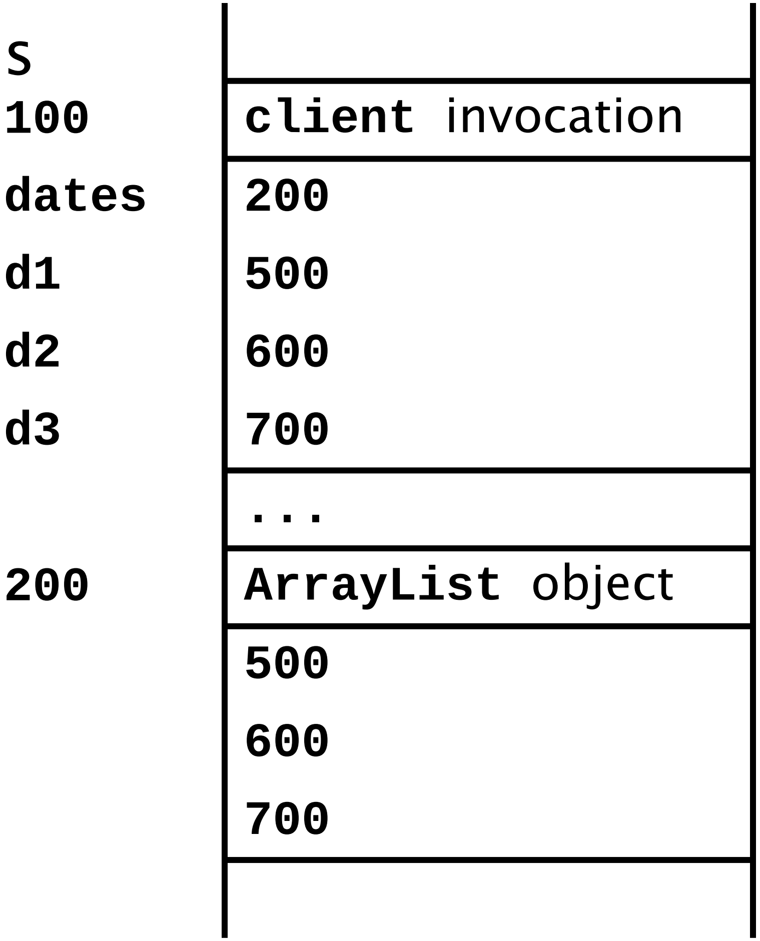
Collections as Attributes

- Often you will want to implement a class that has-a collection as an attribute
  - A university has-a collection of faculties and each faculty has-a collection of schools and departments
  - A molecule has-a collection of atoms
  - A person has-a collection of acquaintances
  - A student has-a collection of GPAs and has-a collection of courses
  - A polygonal model has-a collection of triangles

What Does a Collection Hold?

- A collection holds references to instances
  - It does not hold the instances

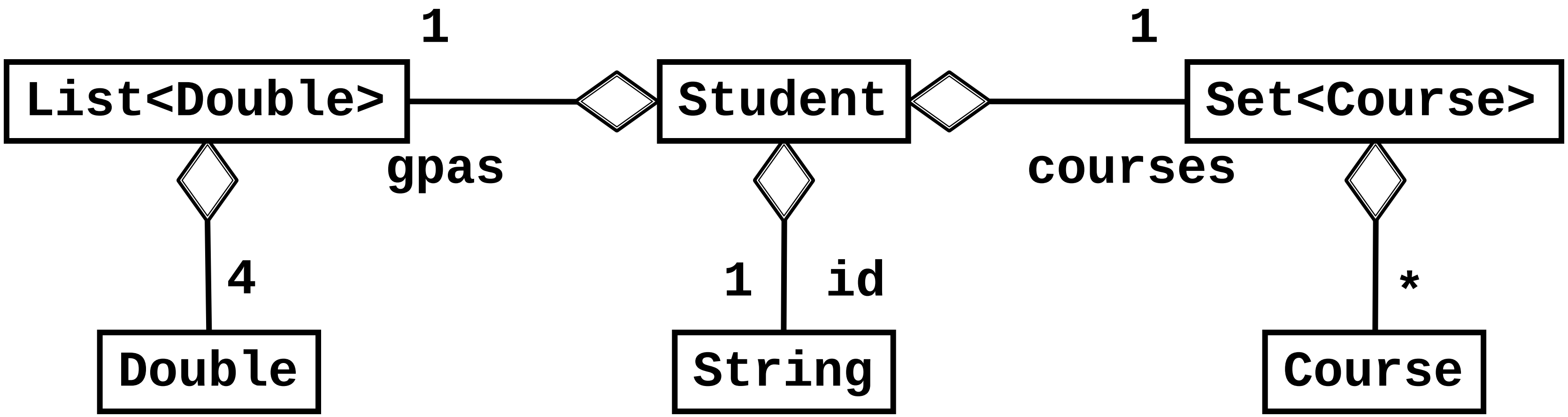
```
ArrayList<Date> dates =
        new ArrayList<Date>();

Date d1 = new Date();
Date d2 = new Date();
Date d3 = new Date();

dates.add(d1);
dates.add(d2);
dates.add(d3);
```



100	client invocation
dates	200
d1	500
d2	600
d3	700
	...
200	ArrayList object
	500
	600
	700

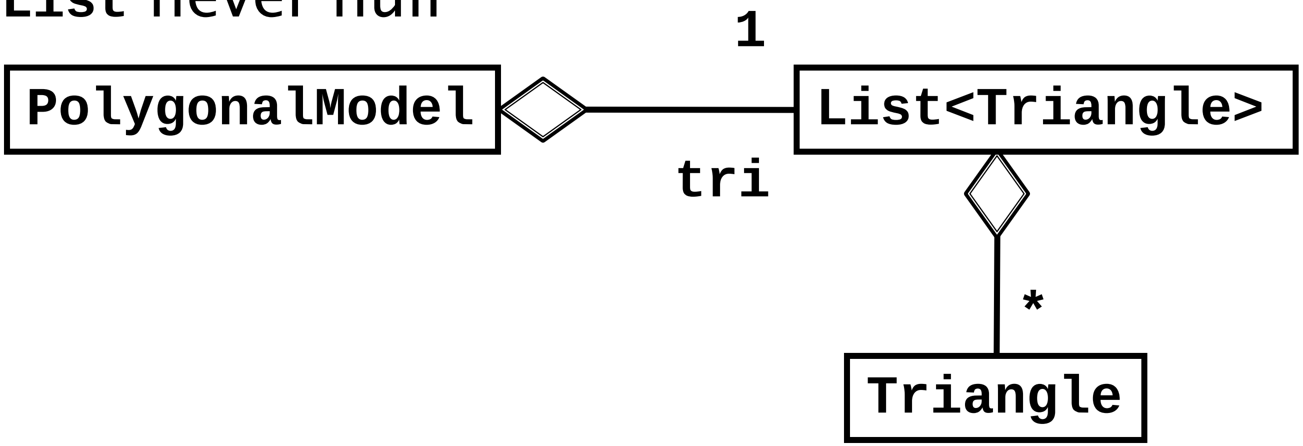
Student Class

- A Student has-a string id
- A Student has-a collection of yearly GPAs
- A Student has-a collection of courses



PolygonalModel Class

- A polygonal model has-a **`List`** of **`Triangles`**
  - Aggregation
- Implements **`Iterable<Triangle>`**
  - Allows clients to access each **`Triangle`** sequentially
- Class invariant
  - **`List`** never null



PolygonalModel

```
class PolygonalModel implements Iterable<Triangle>
{
  private List<Triangle> tri;

  public PolygonalModel()
  {
    tri = new ArrayList<Triangle>();
  }

  public Iterator<Triangle> iterator()
  {
    return this.tri.iterator();
  }
```

PolygonalModel

```
public void clear()
{
  // removes all Triangles
  this.tri.clear();
}

public int size()
{
  // returns the number of Triangles
  return this.tri.size();
}
```

Collections as Attributes

- When using a collection as an attribute of a class **X** you need to decide on ownership issues
  - Does **X** own or share its collection?
  - If **X** owns the collection, does **X** own the objects held in the collection?

x Shares its Collection with other xs

- If **X** shares its collection with other **X** instances, then the copy constructor does not need to create a new collection
  - The copy constructor can simply assign its collection
  - The text refer to this as aliasing

PolygonalModel Copy Constructor 1

```
public PolygonalModel(PolygonalModel p)
{
  // implements aliasing (sharing) with other
  //   PolygonalModel instances
  this.setTriangles( p.getTriangles() );
}

private List<Triangle> getTriangles()
{ return this.tri; }

private void setTriangles(List<Triangle> tri)
{ this.tri = tri; }
```

alias: no new **`List`** created

X Owns its Collection: Shallow Copy

- If **X** owns its collection but not the objects in the collection then the copy constructor can perform a shallow copy of the collection
- A shallow copy of a collection means
  - **X** creates a new collection
  - The references in the collection are aliases for references in the other collection

X Owns its Collection: Shallow Copy

- The hard way to perform a shallow copy

```
// assume there is an ArrayList<Date> dates
ArrayList<Date> sCopy = new ArrayList<Date>();
for(Date d : dates)
{
  sCopy.add(d);
}
```

add does not create new objects

shallow copy: new **List** created but elements are all aliases

X Owns its Collection: Shallow Copy

- The easy way to perform a shallow copy

```
// assume there is an ArrayList<Date> dates
ArrayList<Date> sCopy = new ArrayList<Date>(dates);
```

X Owns its Collection: Deep Copy

- If **X** owns its collection and the objects in the collection then the copy constructor must perform a deep copy of the collection
- A deep copy of a collection means
  - **X** creates a new collection
  - The references in the collection are references to new objects (that are copies of the objects in other collection)

X Owns its Collection: Deep Copy

- How to perform a deep copy

```
// assume there is an ArrayList<Date> dates
ArrayList<Date> sCopy = new ArrayList<Date>();
for(Date d : dates)
{
  sCopy.add(new Date(d.getTime()));
}
```

deep copy: new **`List`** created and new elements created

constructor invocation creates a new object