Mixing Static and Non-static

Singleton

Singleton Pattern

- “There can be only one.”

- Connor MacLeod, Highlander

Singleton Pattern

- a singleton is a class that is instantiated exactly once
- singleton is a well-known design pattern that can be used when you need to:
  1. ensure that there is one, and only one*, instance of a class, and
  2. provide a global point of access to the instance
     - any client that imports the package containing the singleton class can access the instance

[notes 3.4]

*or possibly zero

One and Only One

- how do you enforce this?
  - need to prevent clients from creating instances of the singleton class
    - **`private`** constructors
  - the singleton class should create the one instance of itself
    - note that the singleton class is allowed to call its own **`private`** constructors
    - need a **`static`** attribute to hold the instance

A Silly Example: Version 1

uses a public field that all clients can access

```
package xmas;

public class Santa
{
  // whatever fields you want for santa...

  public static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize attributes here... }

}
```

```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.INSTANCE.givePresent();
}
```

A Silly Example: Version 2

uses a private field; how do clients access the field?

```
package xmas;

public class Santa
{
  // whatever fields you want for santa...

  private static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize attributes here... }

}
```

Global Access

- how do clients access the singleton instance?
  - by using a static method

- note that clients only need to import the package containing the singleton class to get access to the singleton instance
  - any client method can use the singleton instance without mentioning the singleton in the parameter list

A Silly Example (cont)

```
package xmas;

public class Santa {
  private int numPresents;
  private static final Santa INSTANCE = new Santa();

  private Santa()
  { // initialize fields here... }

  public static Santa getInstance()
  { return Santa.INSTANCE; }

  public Present givePresent() {
    Present p = new Present();
    this.numPresents--;
    return p;
  }
}
```

uses a private field; how do clients access the field?

clients use a public static factory method

```java
import xmas;

// client code in a method somewhere ...
public void gimme()
{
  Santa.getInstance().givePresent();
}
```

Enumerations

- an enumeration is a special data type that enables for a variable to be a set of predefined constants
- the variable must be equal to one of the values that have been predefined for it
  - e.g., compass directions
    - NORTH, SOUTH, EAST, and WEST
  - days of the week
    - MONDAY, TUESDAY, WEDNESDAY, etc.
  - playing card suits
    - CLUBS, DIAMONDS, HEARTS, SPADES
- useful when you have a fixed set of constants

A Silly Example: Version 3

```
package xmas;

public enum Santa
{
  // whatever fields you want for santa...

  INSTANCE;

  private Santa()
  { // initialize attributes here... }

}
```

singleton as an enumeration

will call the private default constructor same usage as public field (Version 1)

```
import xmas;

// client code in a method somewhere ...
public void gimme()
{
   Santa.INSTANCE.givePresent();
}
```

Singleton as an enumeration

- considered the preferred approach for implementing a singleton
  - for reasons beyond the scope of CSE1030

- all enumerations are subclasses of **`java.lang.Enum`**

Applications

- singletons should be uncommon
- typically used to represent a system component that is intrinsically unique
  - window manager
  - file system
  - logging system

Logging

- when developing a software program it is often useful to log information about the runtime state of your program
  - similar to flight data recorder in an airplane
  - a good log can help you find out what went wrong in your program
- problem: your program may have many classes, each of which needs to know where the single logging object is
  - global point of access to a single object == singleton
- Java logging API is more sophisticated than this
  - but it still uses a singleton to manage logging
  - java.util.logging

Lazy Instantiation

- notice that the previous singleton implementation always creates the singleton instance whenever the class is loaded
  - if no client uses the instance then it was created needlessly
- it is possible to delay creation of the singleton instance until it is needed by using lazy instantiation
  - only works for version 2

Lazy Instantiation as per Notes

```
public class Santa {
  private static Santa INSTANCE = null;

  private Santa()
  { // ... }

  public static Santa getInstance()
  {
    if (Santa.INSTANCE == null) {
      Santa.INSTANCE = new Santa();
    }
    return Santa.INSTANCE;
  }
}
```

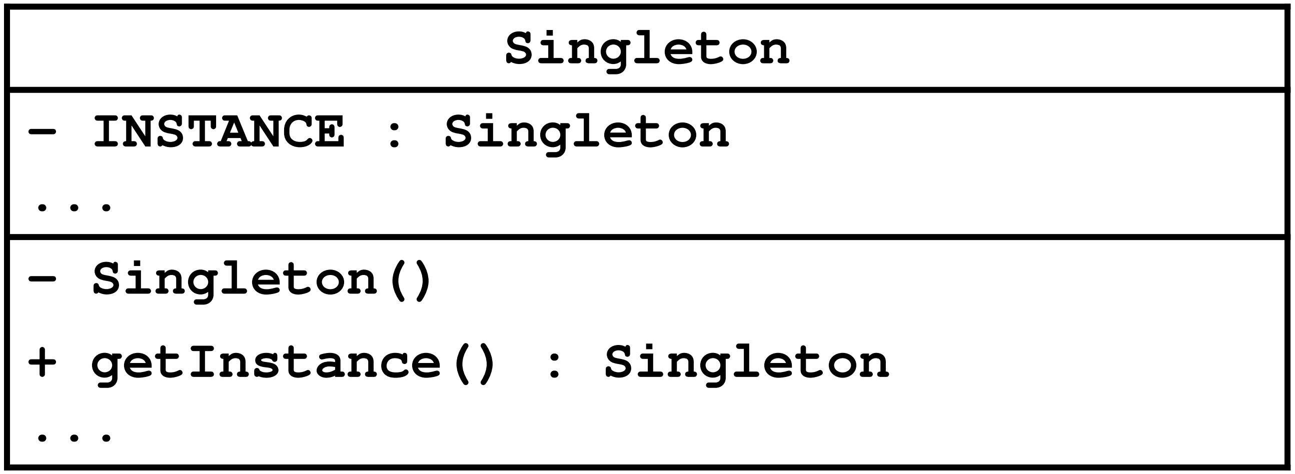
Mixing Static and Non-static

Multiton

Goals for Today

- Multiton
- review maps
- static factory methods

Singleton UML Class Diagram



Singleton

− INSTANCE : Singleton

...

− Singleton()

+ getInstance() : Singleton

...

One Instance per State

- the Java language specification guarantees that identical **`String`** literals are not duplicated

```
// client code somewhere

String s1 = "xyz";
String s2 = "xyz";

// how many String instances are there?
System.out.println("same object? " + (s1 == s2) );
```

  - prints: **`same object? true`**
- the compiler ensures that identical **`String`** literals all refer to the same object
  - a single instance per unique state

[notes 3.5]

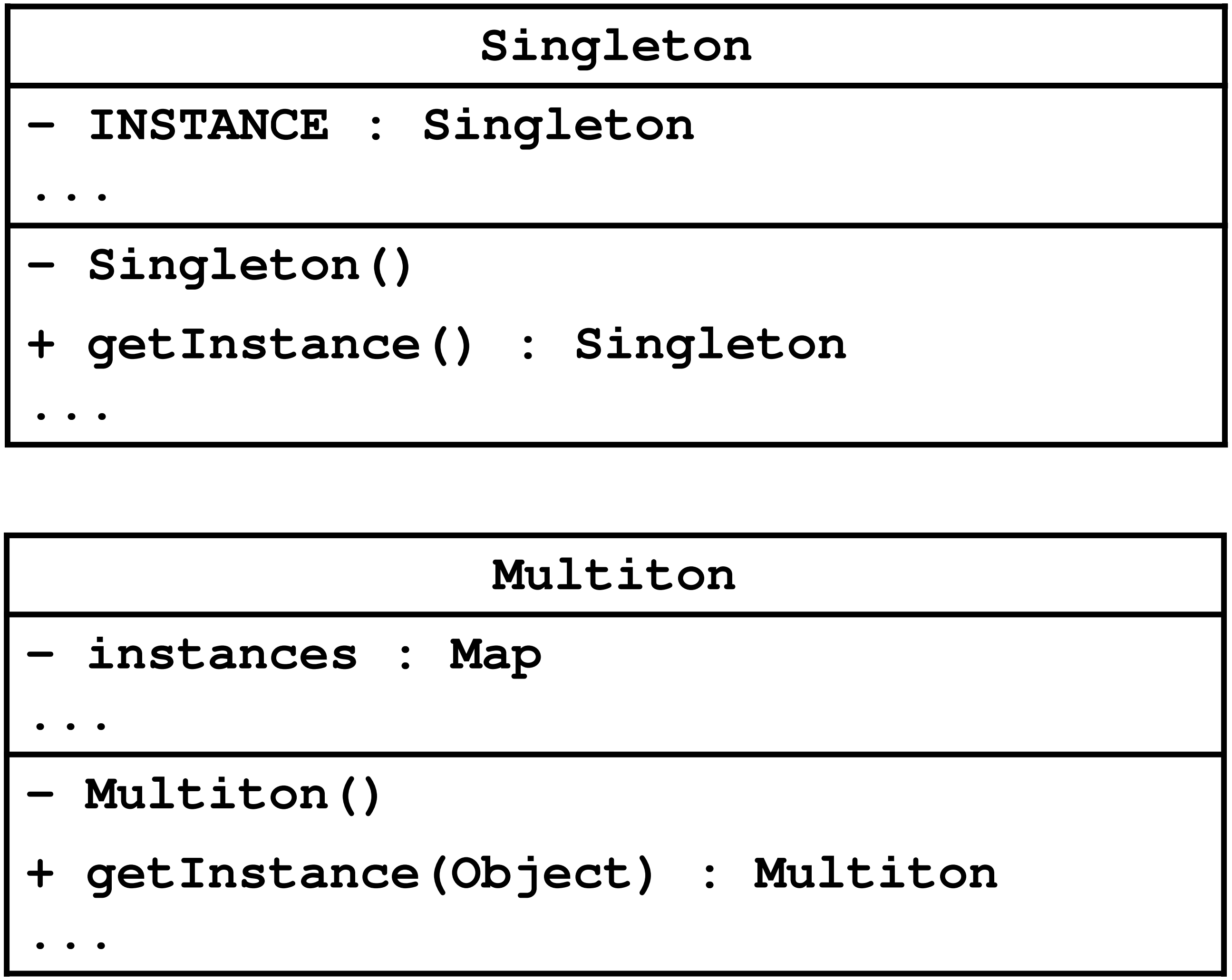
Multiton

- a *singleton* class manages a single instance of the class
- a *multiton* class manages multiple instances of the class

- what do you need to manage multiple instances?
  - a collection of some sort

- how does the client request an instance with a particular state?
  - it needs to pass the desired state as arguments to a method

Singleton vs Multiton UML Diagram



```
Singleton
- INSTANCE : Singleton
...
- Singleton()
+ getInstance() : Singleton
...

Multiton
- instances : Map
...
- Multiton()
+ getInstance(Object) : Multiton
...
```

Singleton vs Multiton

- Singleton
  - one instance

```
private static final Santa INSTANCE = new Santa();
```

  - zero-parameter accessor

```
public static Santa getInstance()
```

Singleton vs Multiton

- Multiton
  - multiple instances (each with unique state)

```
private static final Map<String, PhoneNumber>
  instances = new TreeMap<String, PhoneNumber>();
```

  - accessor needs to provide state information

```
public static PhoneNumber getInstance(int areaCode,
                                    int exchangeCode,
                                    int stationCode)
```

Map

- a map stores key-value pairs

```
Map<String, PhoneNumber>
```

key type value type

- values are put into the map using the key

```
// client code somewhere
Map<String, PhoneNumber> m =
                      new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648"

m.put(key, ago);
```

[AJ 16.2]

- values can be retrieved from the map using only the key
  - if the key is not in the map the value returned is `null`

```java
// client code somewhere
Map<String, PhoneNumber> m =
                      new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648";

m.put(key, ago);

PhoneNumber gallery = m.get(key);            // == ago
PhoneNumber art = m.get("4169796648");       // == ago

PhoneNumber pizza = m.get("4169671111");     // == null
```

- a map is not allowed to hold duplicate keys
  - if you re-use a key to insert a new object, the existing object corresponding to the key is removed and the new object inserted

```
// client code somewhere
Map<String, PhoneNumber> m = new TreeMap<String, PhoneNumber>;

PhoneNumber ago = new PhoneNumber(416, 979, 6648);
String key = "4169796648";

m.put(key, ago);                              // add ago
System.out.println(m);

m.put(key, new PhoneNumber(905, 760, 1911));  // replaces ago
System.out.println(m);
```

prints

```
{4169796648=(416) 979-6648}
{4169796648=(905) 760-1911}
```

Mutable Keys

- from
  http://docs.oracle.com/javase/7/docs/api/java/util/Map.html
  - Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map.

```
public class MutableKey
{
  public static void main(String[] args)
  {
    Map<Date, String> m = new TreeMap<Date, String>();
    Date d1 = new Date(100, 0, 1);
    Date d2 = new Date(100, 0, 2);
    Date d3 = new Date(100, 0, 3);
    m.put(d1, "Jan 1, 2000");
    m.put(d2, "Jan 2, 2000");
    m.put(d3, "Jan 3, 2000");
    d2.setYear(101);           // mutator
    System.out.println("d1 " + m.get(d1));  // d1 Jan 1, 2000
    System.out.println("d2 " + m.get(d2));  // d2 Jan 2, 2000
    System.out.println("d3 " + m.get(d3));  // d3 null
  }
}
```

don't mutate keys;
bad things will happen

change TreeMap to HashMap and see what happens

Making **`PhoneNumber`** a Multiton

1. multiple instances (each with unique state)

```
private static final Map<String, PhoneNumber>
    instances = new TreeMap<String, PhoneNumber>();
```

2. accessor needs to provide state information

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
```

- **`getInstance()`** will get an instance from **`instances`** if the instance is in the map; otherwise, it will create the new instance and put it in the map

Making **`PhoneNumber`** a Multiton

3. require private constructors
   - to prevent clients from creating instances on their own
     - clients should use **`getInstance()`**

4. require immutability of **`PhoneNumber`**s
   - to prevent clients from modifying state, thus making the keys inconsistent with the **`PhoneNumber`**s stored in the map
   - recall the recipe for immutability...

```
public class PhoneNumber implements Comparable<PhoneNumber>
{
  private static final Map<String, PhoneNumber> instances =
                         new TreeMap<String, PhoneNumber>();

  private final short areaCode;
  private final short exchangeCode;
  private final short stationCode;

  private PhoneNumber(int areaCode,
                      int exchangeCode,
                      int stationCode)
  { // identical to previous versions }
```

why is validation not needed?

```
public static PhoneNumber getInstance(int areaCode,
                                      int exchangeCode,
                                      int stationCode)
{
  String key = "" + areaCode + exchangeCode + stationCode;
  PhoneNumber n = PhoneNumber.instances.get(key);
  if (n == null)
  {
    n = new PhoneNumber(areaCode, exchangeCode, stationCode);
    PhoneNumber.instances.put(key, n);
  }
  return n;
}
// remainder of PhoneNumber class ...
```

```
public class PhoneNumberClient {

  public static void main(String[] args)
  {
    PhoneNumber x = PhoneNumber.getInstance(416, 736, 2100);
    PhoneNumber y = PhoneNumber.getInstance(416, 736, 2100);
    PhoneNumber z = PhoneNumber.getInstance(905, 867, 5309);

    System.out.println("x equals y: " + x.equals(y) +
                      " and x == y: " + (x == y));

    System.out.println("x equals z: " + x.equals(z) +
                      " and x == z: " + (x == z));
  }
}
```

```
x equals y: true and x == y: true
x equals z: false and x == z: false
```

Bonus Content

- notice that Singleton and Multiton use a static method to return an instance of a class
- a static method that returns an instance of a class is called a *static factory method*
  - factory because, as far as the client is concerned, the method creates an instance
    - similar to a constructor

Static Factory Methods

- many examples
  - `java.lang.Integer`

    `public static Integer valueOf(int i)`
    - Returns a **`Integer`** instance representing the specified **`int`** value.

  - `java.util.Arrays`

    `public static int[] copyOf(int[] original, int newLength)`
    - Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

Java API Static Factory Methods

- `java.lang.String`

  `public static String format(String format, Object... args)`

  - Returns a formatted string using the specified format string and arguments.

- `cse1030.math.Complex`

  `public static Complex fromPolar(double mag, double angle)`

  - Returns a reference to a new complex number given its polar form.

- you can give meaningful names to static factory methods (unlike constructors)

```
public class Person {
  private String name;
  private int age;
  private int weight;

  public Person(String name, int age, int weight) { // ... }

  public Person(String name, int age) { // ... }

  public Person(String name, int weight) { // ... }
  // ...
}
```

illegal overload: same signature

```
public class Person {  // modified from PEx's
  // attributes ...

  public Person(String name, int age, int weight) { // ... }

  public static Person withAge(String name, int age) {
    return new Person(name, age, DEFAULT_WEIGHT);
  }

  public static Person withWeight(String name, int weight) {
    return new Person(name, DEFAULT_AGE, weight);
  }
}
```

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: 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

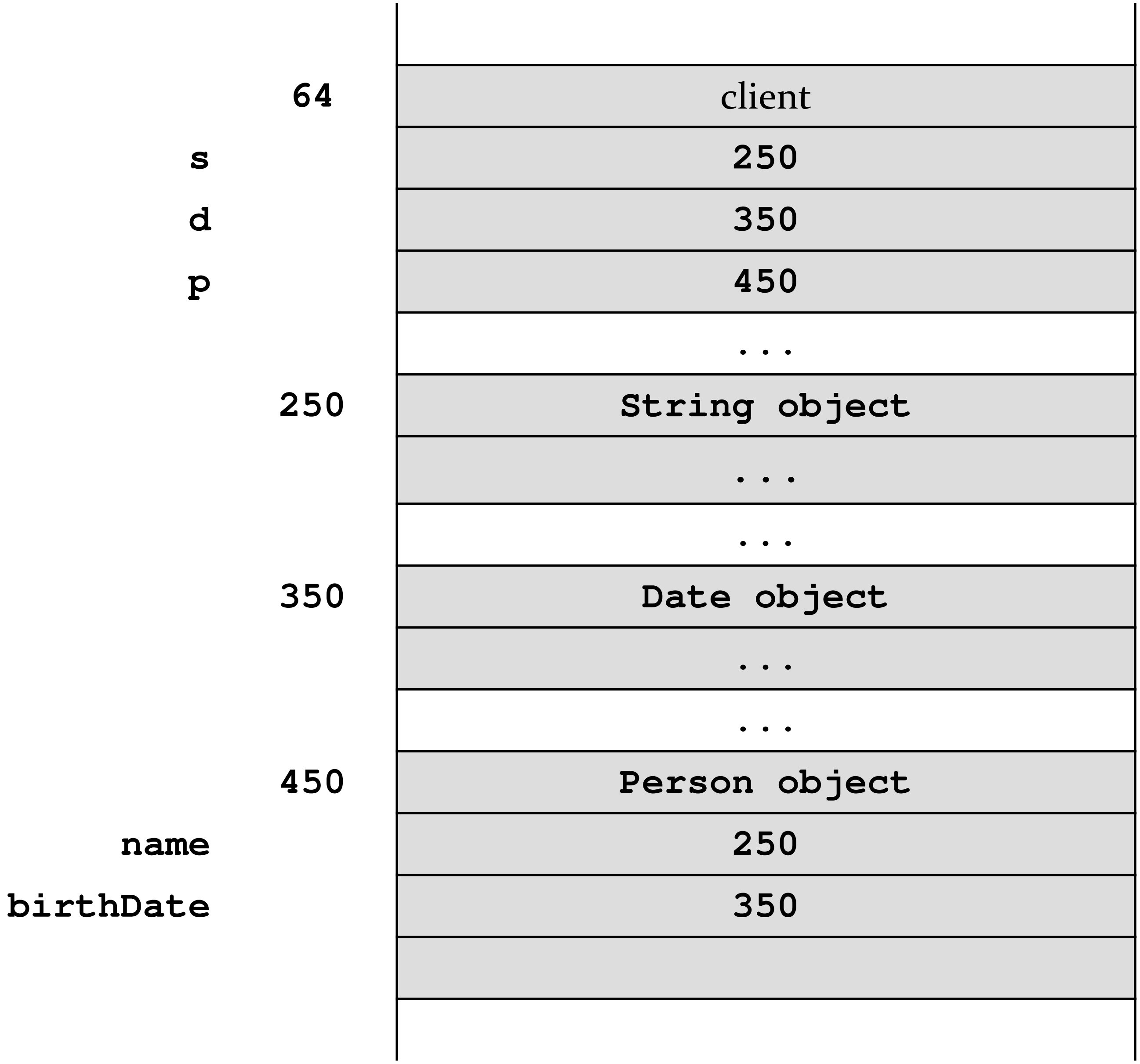
Aggregation

- 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?

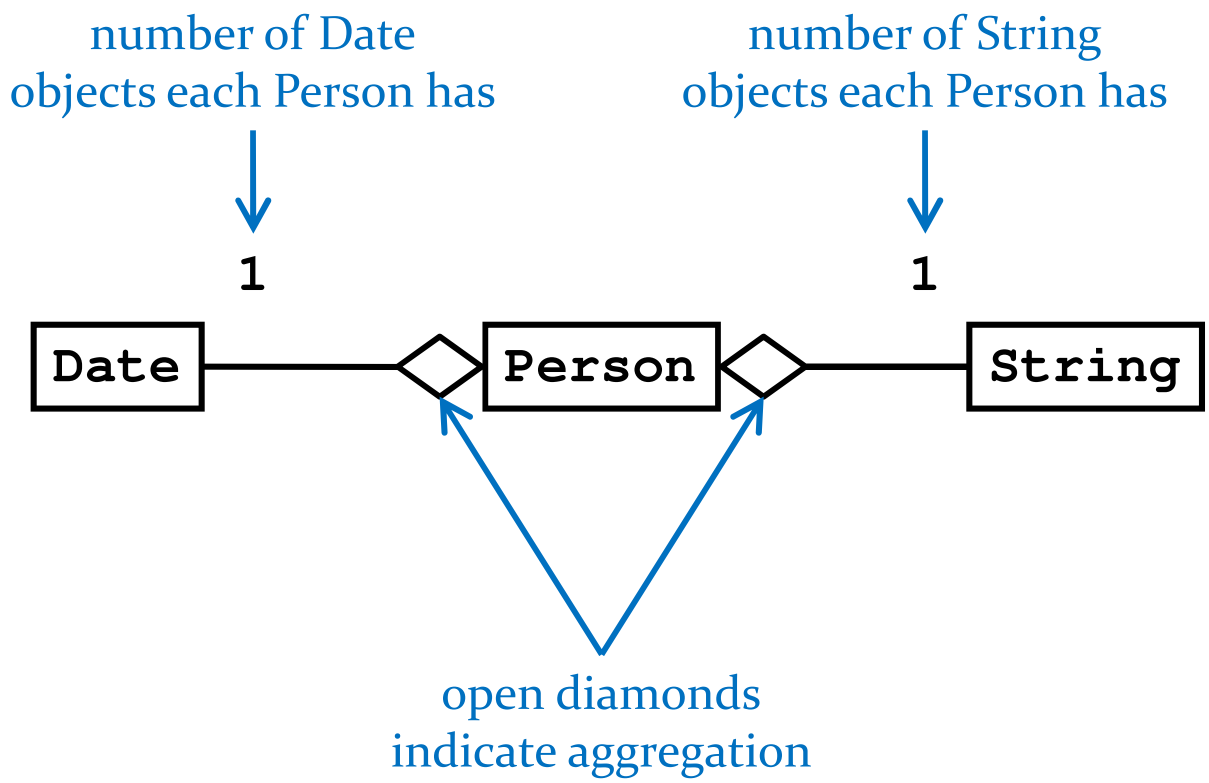
```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(90, 2, 26);   // March 26, 1990
Person p = new Person(s, d);

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`**

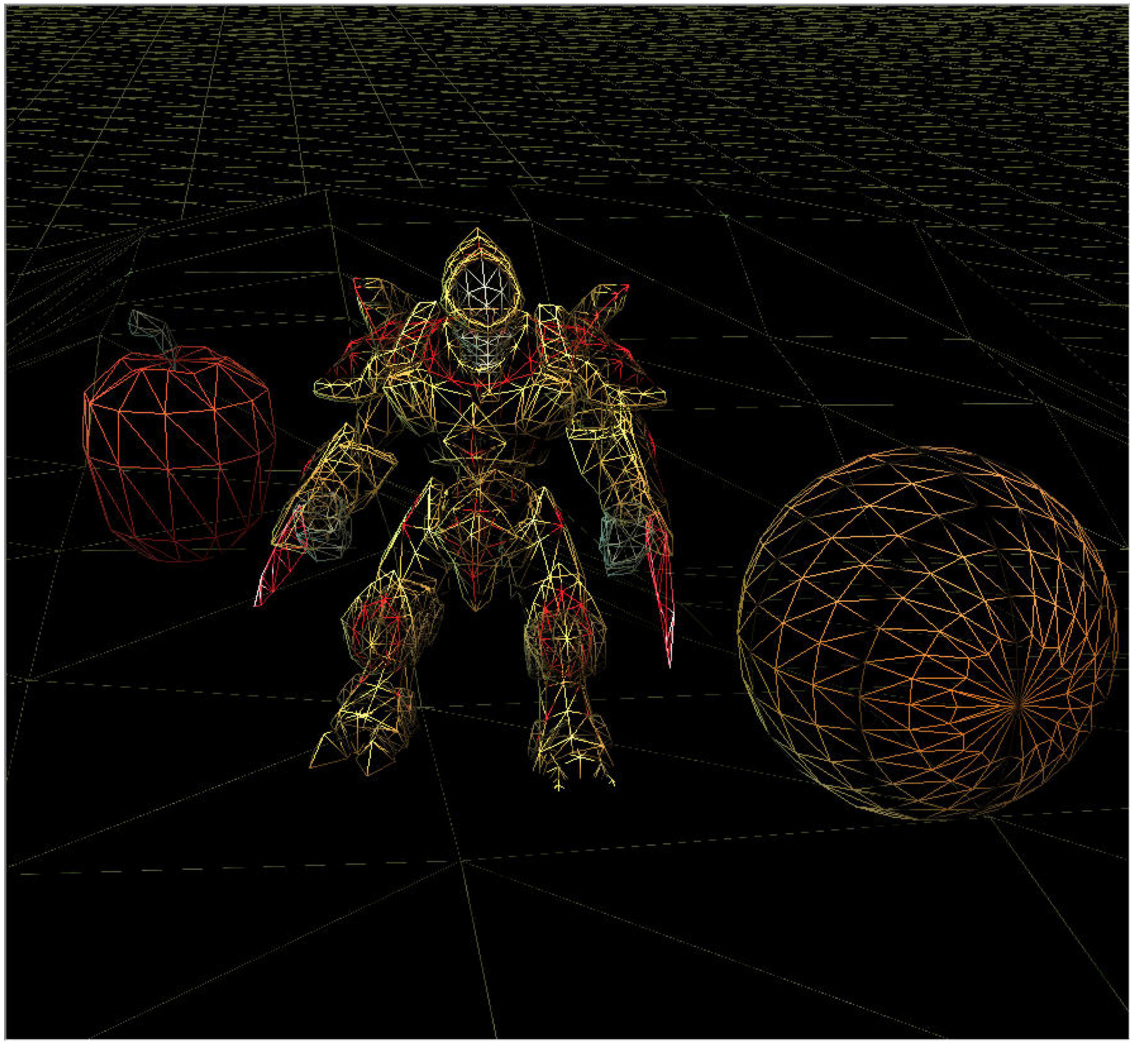
- note that even though the **`String`** instance is shared by the client and the **`Person`** instance **`p`**, neither the client nor **`p`** can modify the **`String`**
  - immutable objects make great building blocks for other objects
  - they can be shared freely without worrying about their state

UML Class Diagram for Aggregation


number of Date objects each Person has
number of String objects each Person has
1
1
Date
Person
String
open diamonds indicate aggregation

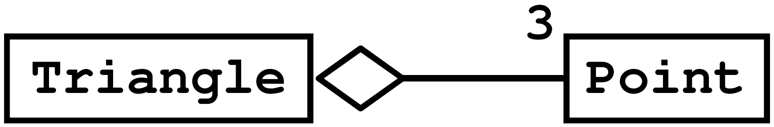
Another Aggregation Example

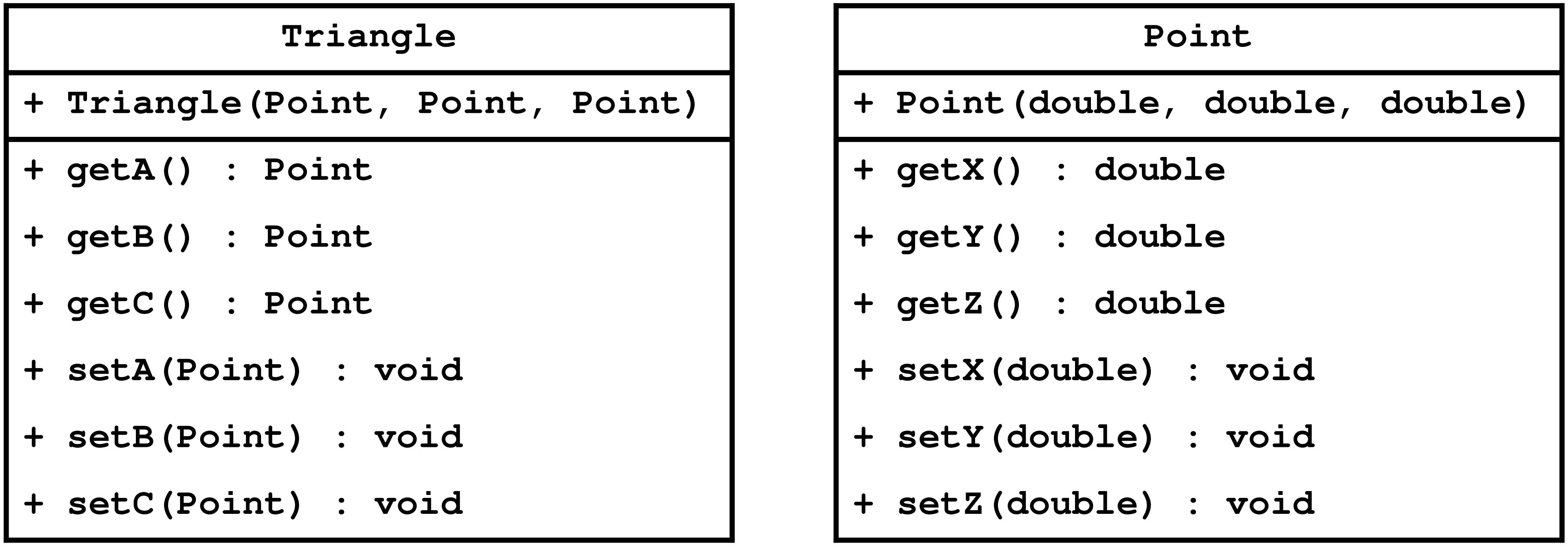
- 3D videogames use models that are a three-dimensional representations of geometric data
  - the models may be represented by:
    - three-dimensional points (particle systems)
    - simple polygons (triangles, quadrilaterals)
    - smooth, continuous surfaces (splines, parametric surfaces)
    - an 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 **`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

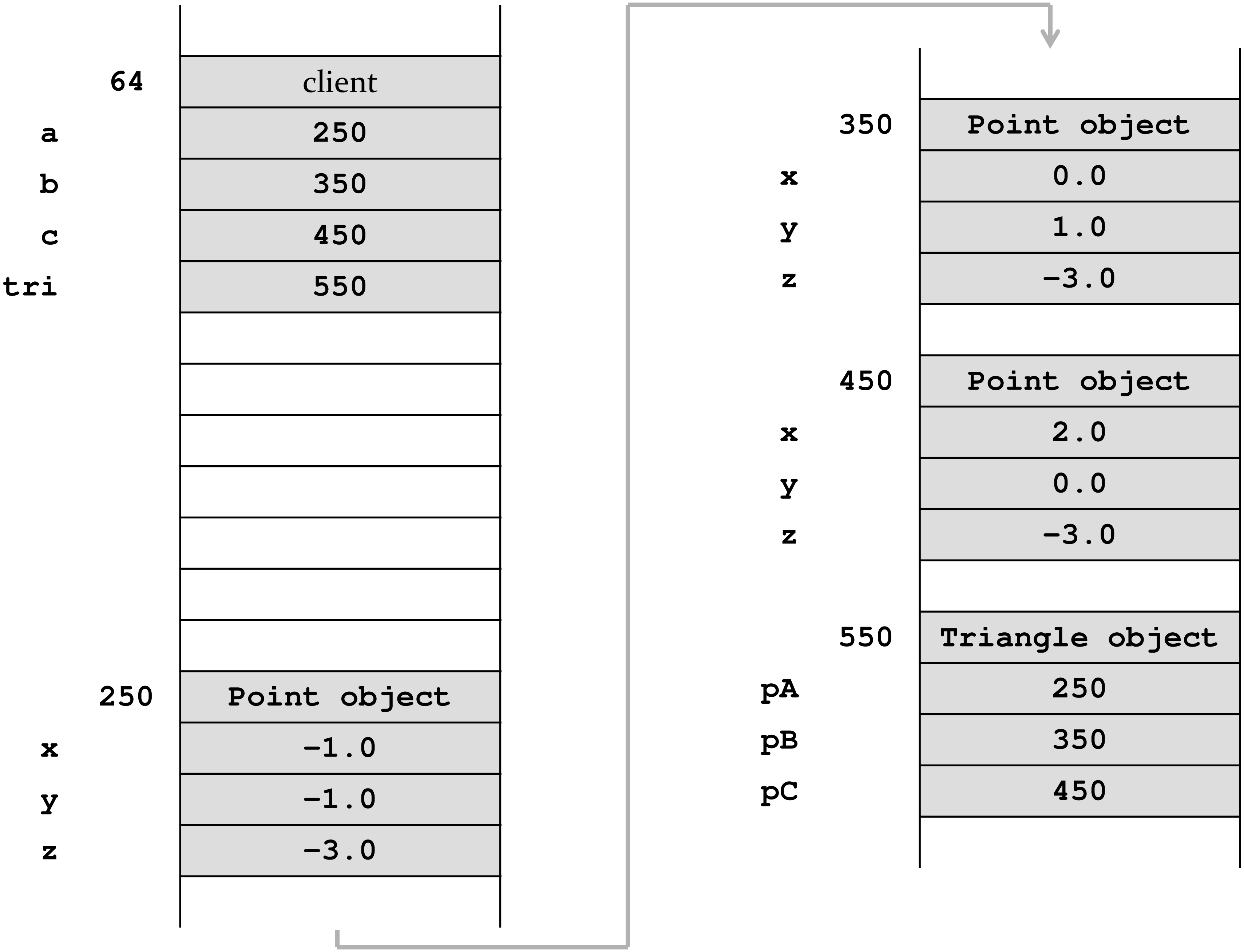
```
  // mutators

  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);
```

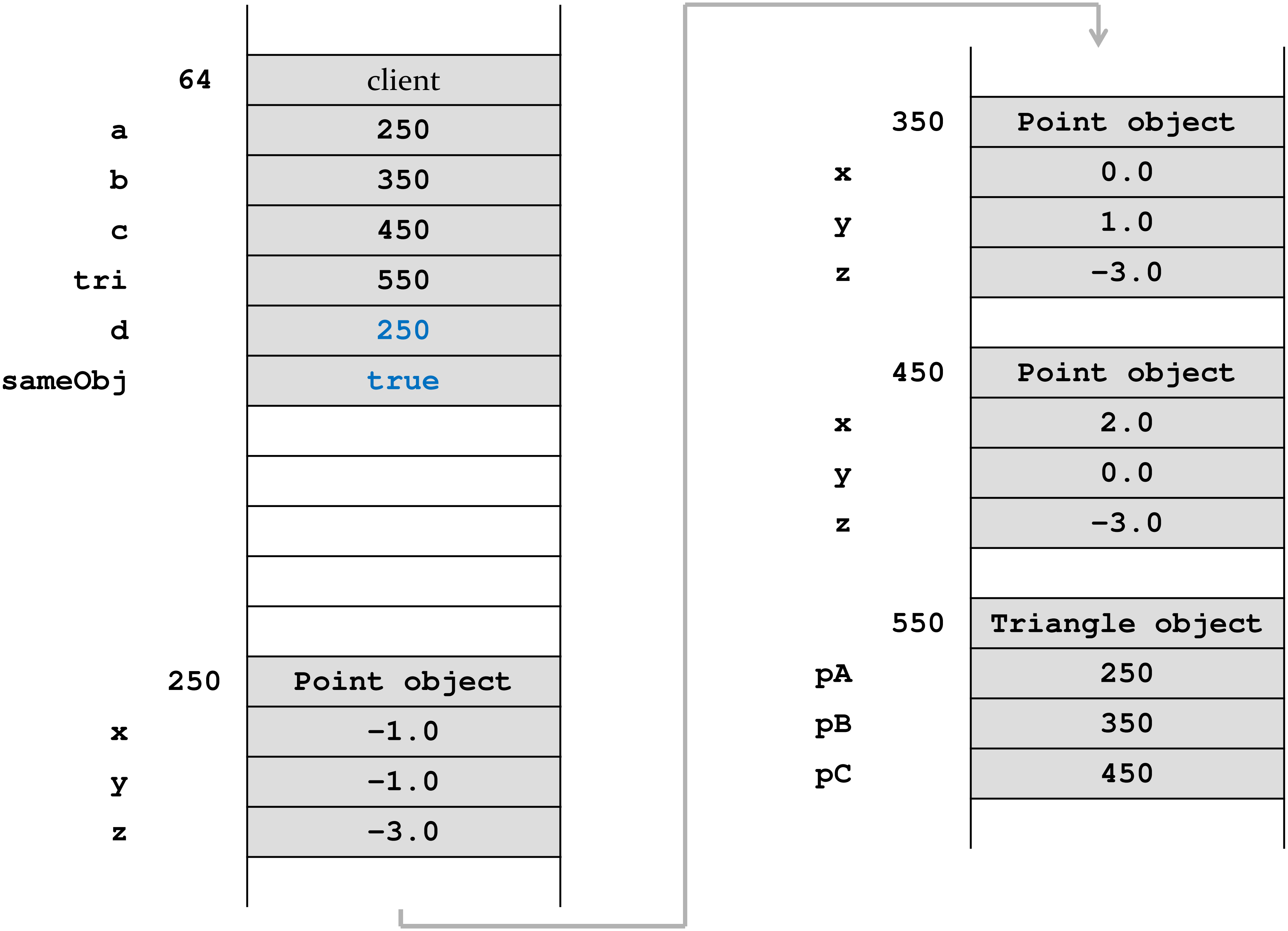


	64	client
a		250
b		350
c		450
tri		550
	250	Point object
x		-1.0
y		-1.0
z		-3.0

	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

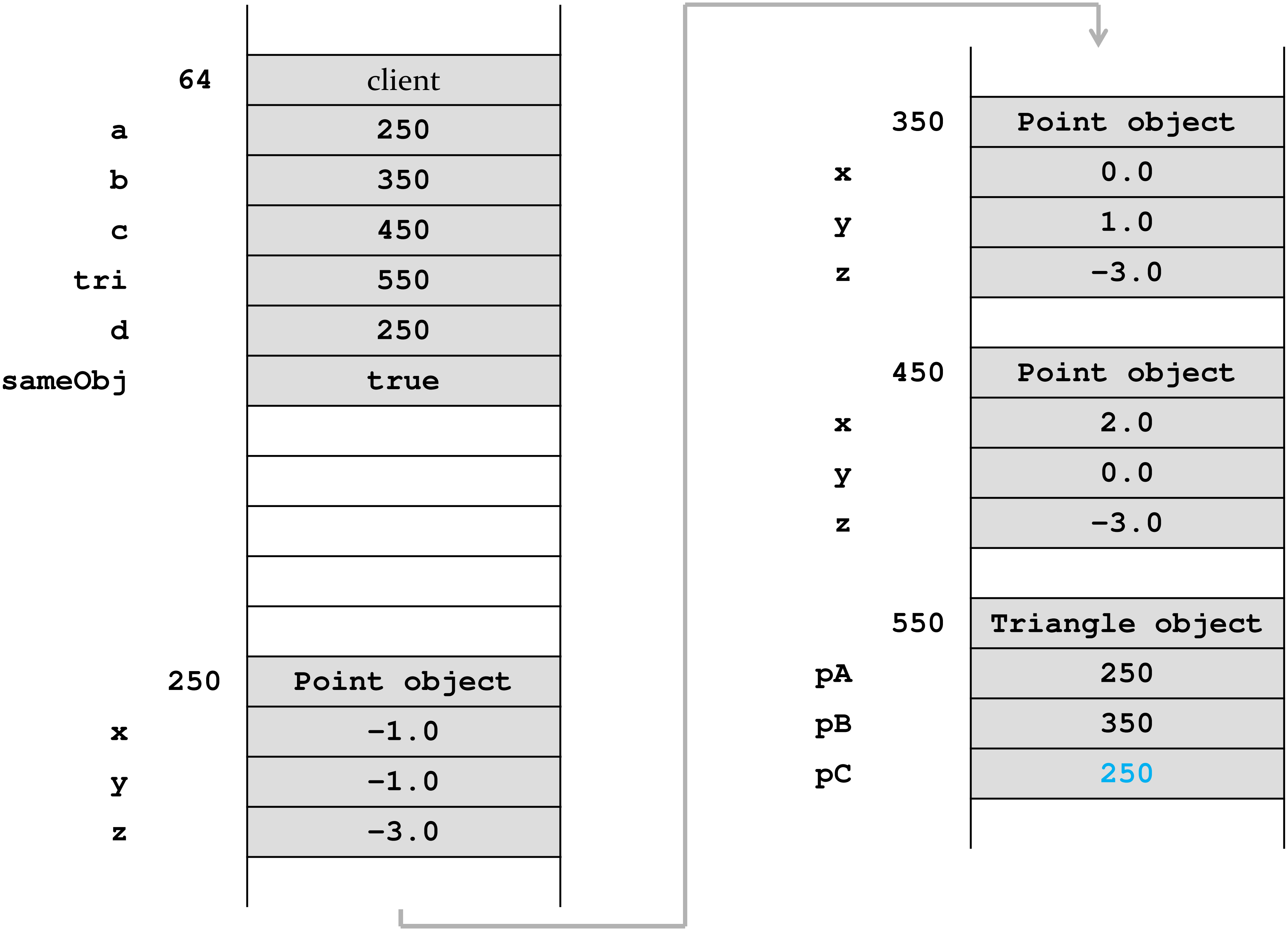


	64	client
a		250
b		350
c		450
tri		550
d		250
sameObj		true
	250	Point object
x		-1.0
y		-1.0
z		-3.0

	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`**

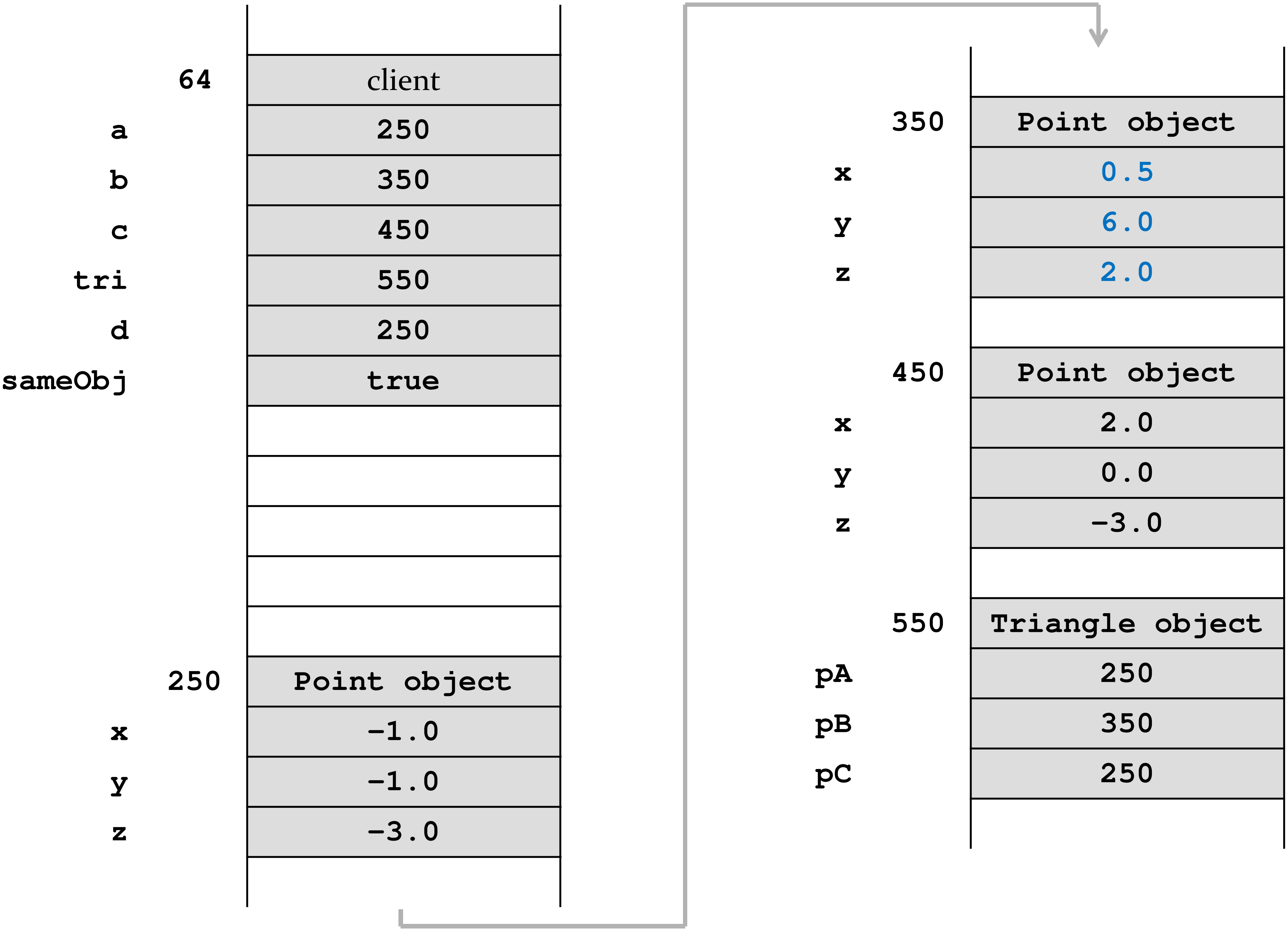


	64	client
a		250
b		350
c		450
tri		550
d		250
sameObj		true
	250	Point object
x		-1.0
y		-1.0
z		-3.0

	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));
```

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

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