EECS 2030 Advanced Object-Oriented Programming

S2023, Section A

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 (as opposed to **is-a**)

- 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

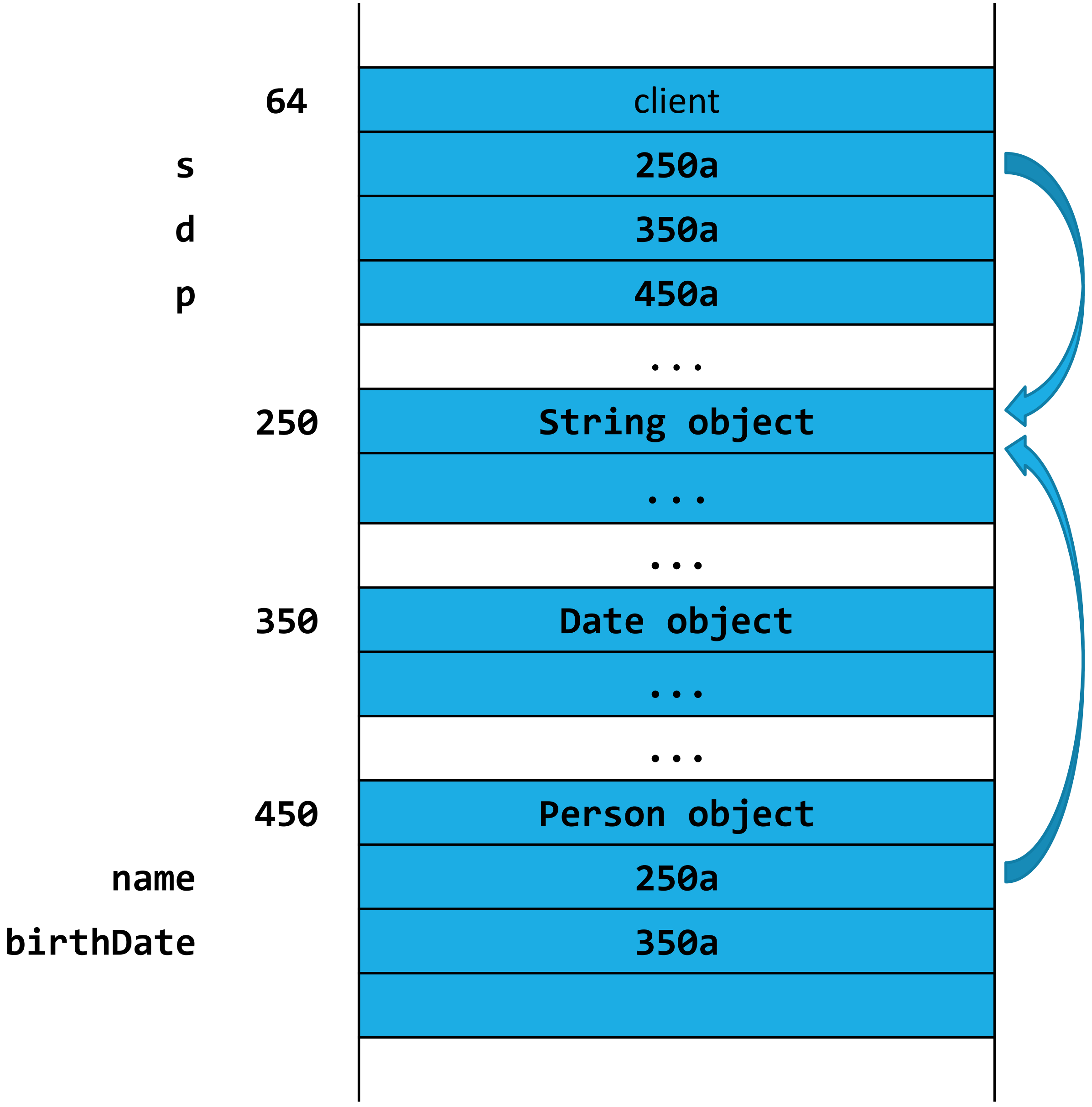
```
public class Person {
  private String name;
  private Date birthDate;
}
```

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

  public String getName() {
    return this.name;
  }

  public Date getBirthDate() {
    return this.birthDate;
  }

}
```

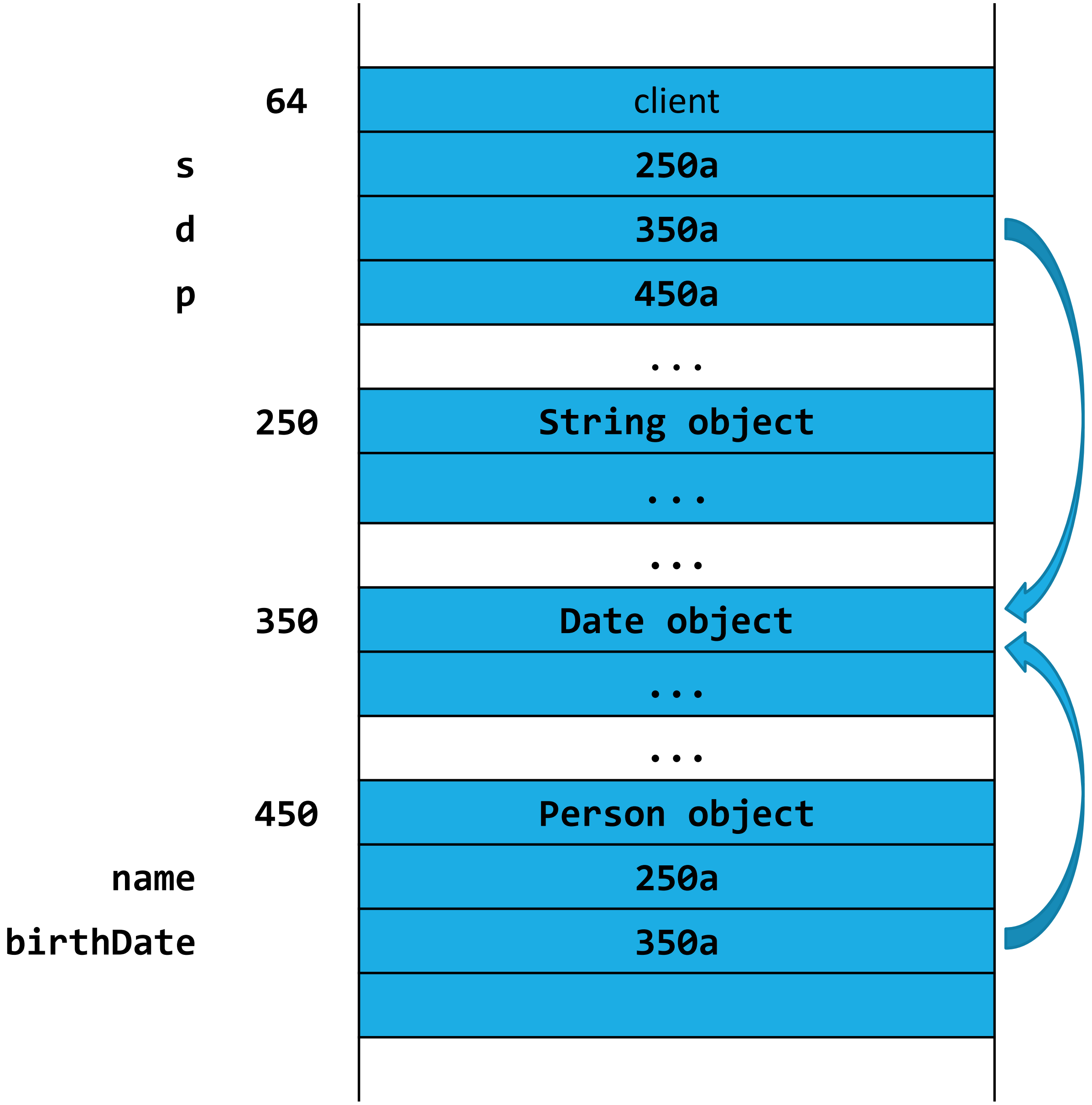
the Person example uses **aggregation**

- Constructor *does not* make a new copy of the name and birth date objects passed to it
- 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);
```



Person object and client have a reference to the same **String** object



Person object and client have a reference to the same **Date** object

Consequences of Sharing

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`

Coupling

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`
- Same with the `Person` instance:
  - `p` can modify the date using `birthDate` and the client sees a modified date `d`

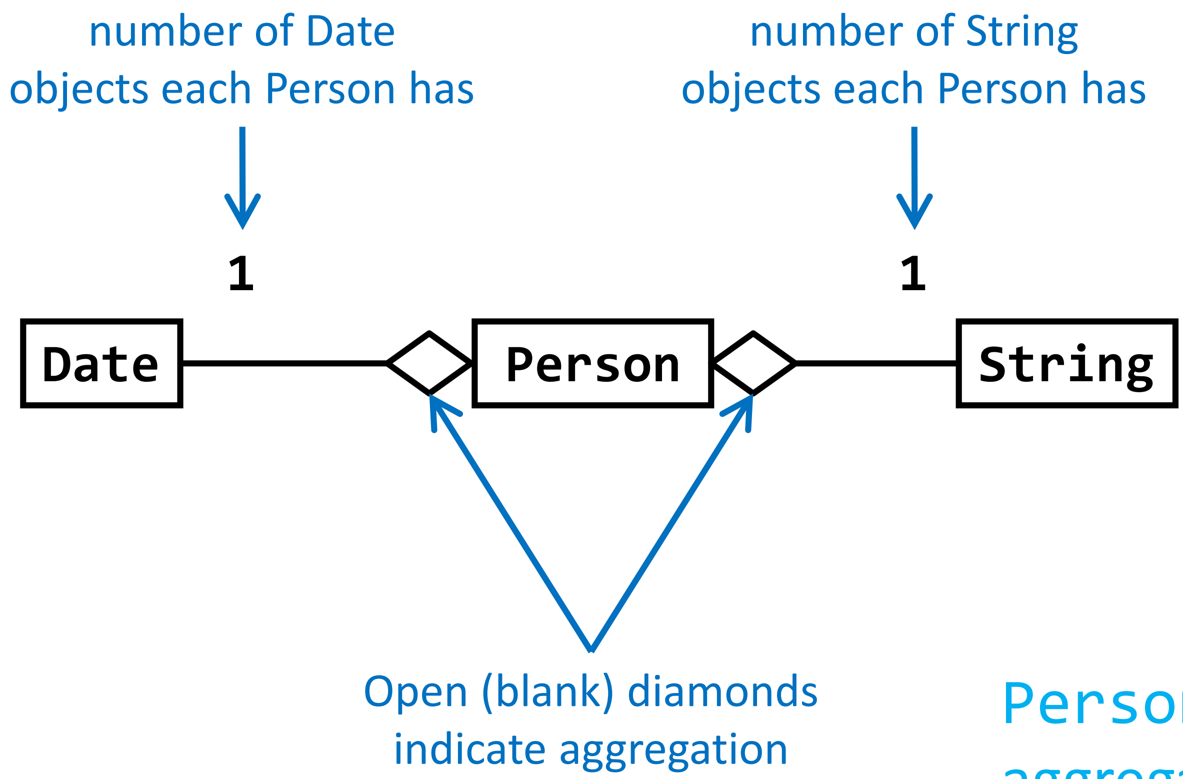
No problem with String

`String` instance is shared by the client and the `Person` instance `p`

BUT neither the client nor `p` can modify it

- immutable objects make great building blocks for other objects
- 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 (blank) diamonds indicate aggregation

Person is an aggregation of Date and String

Another Aggregation Example

consider implementing a projectile whose position is governed by the following equations of motion

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{v}_i \delta t + \frac{1}{2}\mathbf{g}\delta t^2$$

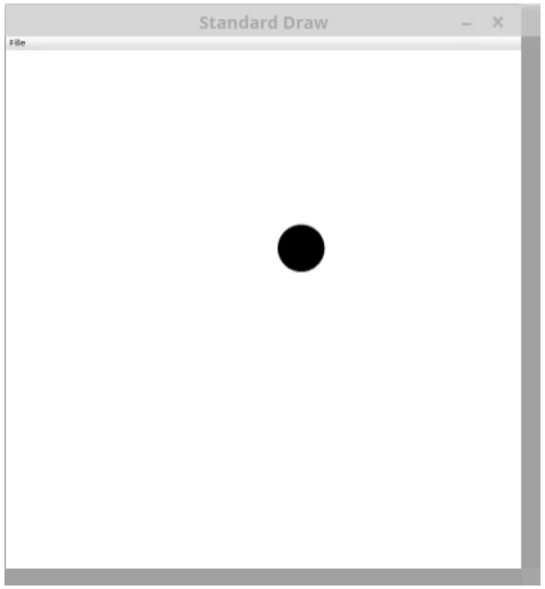
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{g}\delta t$$

$\mathbf{p}_i$ position at time t_i

$\mathbf{v}_i$ velocity at time t_i

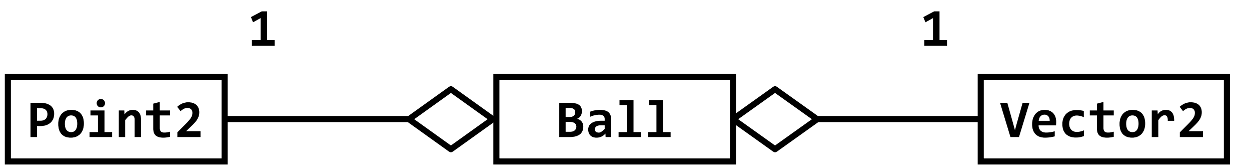
$\mathbf{g}$ acceleration due to gravity

$\delta t = t_{i+1} - t_i$



Another Aggregation Example

the **Projectile** has-a **Point2** that represents the position of the ball and a **Vector2** that represents the velocity of the ball



Ball is an aggregation of Point2 and Vector2

```
/**
 * A particle moving with approximate projectile motion in two
 * dimensions. It is assumed that the only force acting on
 * the projectile is gravity.
 */
public class Projectile {

    /**
     * The current position of the projectile.
     */
    private Point2 pos;

    /**
     * The current velocity of the projectile.
     */
    private Vector2 vel;


   /**
     * Initialize this projectile to start at position (0, 0)
     * having velocity (0, 0).
     *
     */
    public Projectile() {
        this.pos = new Point2();
        this.vel = new Vector2();
    }
```

```
  /**
   * Set the position of this projectile to the specified position.
   * Returns the old position of this projectile.
   *
   * @param p the new position of this projectile
   * @return the old position of this projectile
   */
  public Point2 setPosition(Point2 p) {
      Point2 oldPos = this.pos;
      this.pos = p;
      return oldPos;
  }

 /**
   * Set the position of this projectile to the specified velocity.
   * Returns the old velocity of this projectile.
   *
   * @param p the new velocity of this projectile
   * @return the old velocity of this projectile
   */
  public Vector2 setVelocity(Vector2 v) {
      Vector2 oldVel = this.vel;
      this.vel = v;
      return oldVel;
  }
```

```
/**
 * Returns a reference to the position of this projectile.
 *
 * @return a reference to the position of this projectile
 */
public Point2 getPosition() {
    return this.position;
}

/**
 * Returns a reference to the velocity of this projectile.
 *
 * @return a reference to the velocity of this projectile
 */
public Vector2 getVelocity() {
    return this.velocity;
}
```

```java
/**
 * Updates the position and velocity of this projectile after
 * the projectile has moved {@code dt} seconds from its previous
 * position.
 *
 * @param dt the time period over which the projectile has moved
 */
public void move(double dt) {
    // acceleration due to gravity
    Vector2 g = new Vector2();
    g.set(0.0, -9.81);

    this.pos.add(Vector2.multiply(dt, this.vel)).
             add(Vector2.multiply(0.5 * dt * dt, g));
    this.vel.add(Vector2.multiply(dt, g));
}
```

`Projectile` as an aggregation

implementing **`Projectile`** is very easy:

fields

- are references to existing objects provided by the client

accessors

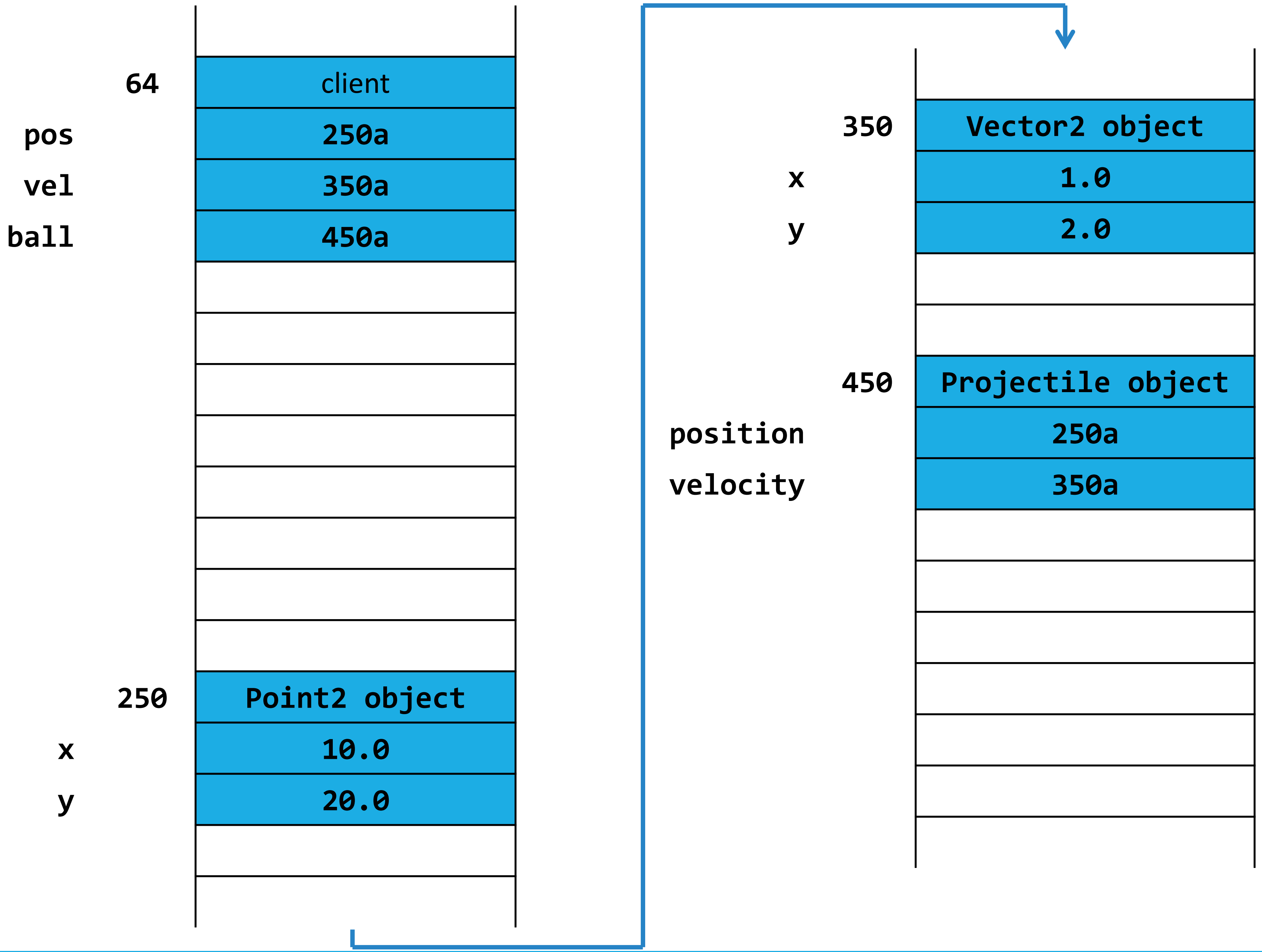
- give clients a reference to the aggregated **`Point2`** and **`Vector2`** objects

mutators

- set fields to existing object references provided by the client

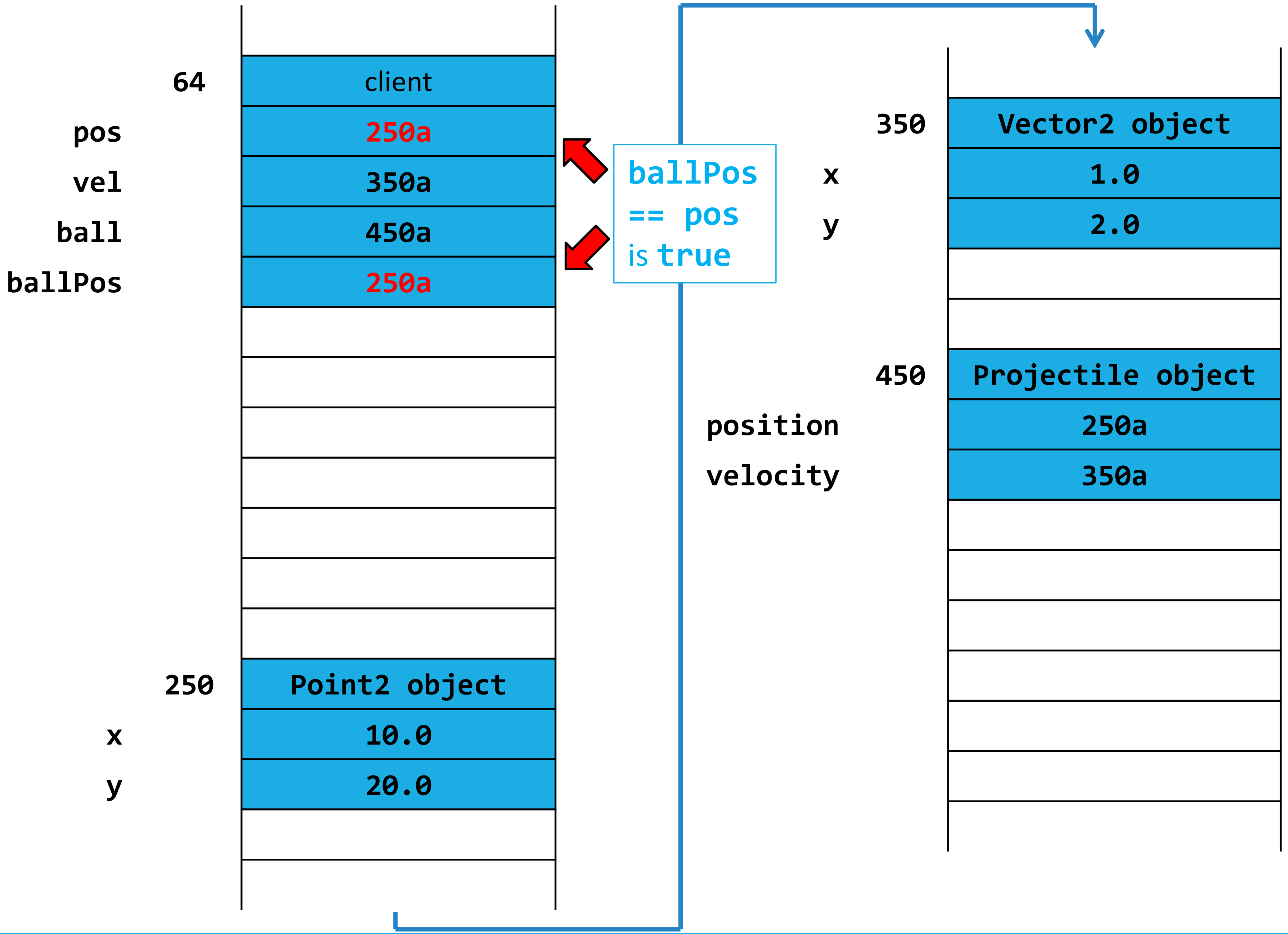
we say that the **`Projectile`** fields are *aliases*

```java
public static void main(String[] args) {
    Point2 pos = new Point2(10.0, 20.0);
    Vector2 vel = new Vector2(1.0, 2.0);
    Projectile ball = new Projectile();
    ball.setPosition(pos);
    ball.setVelocity(vel);
}
```



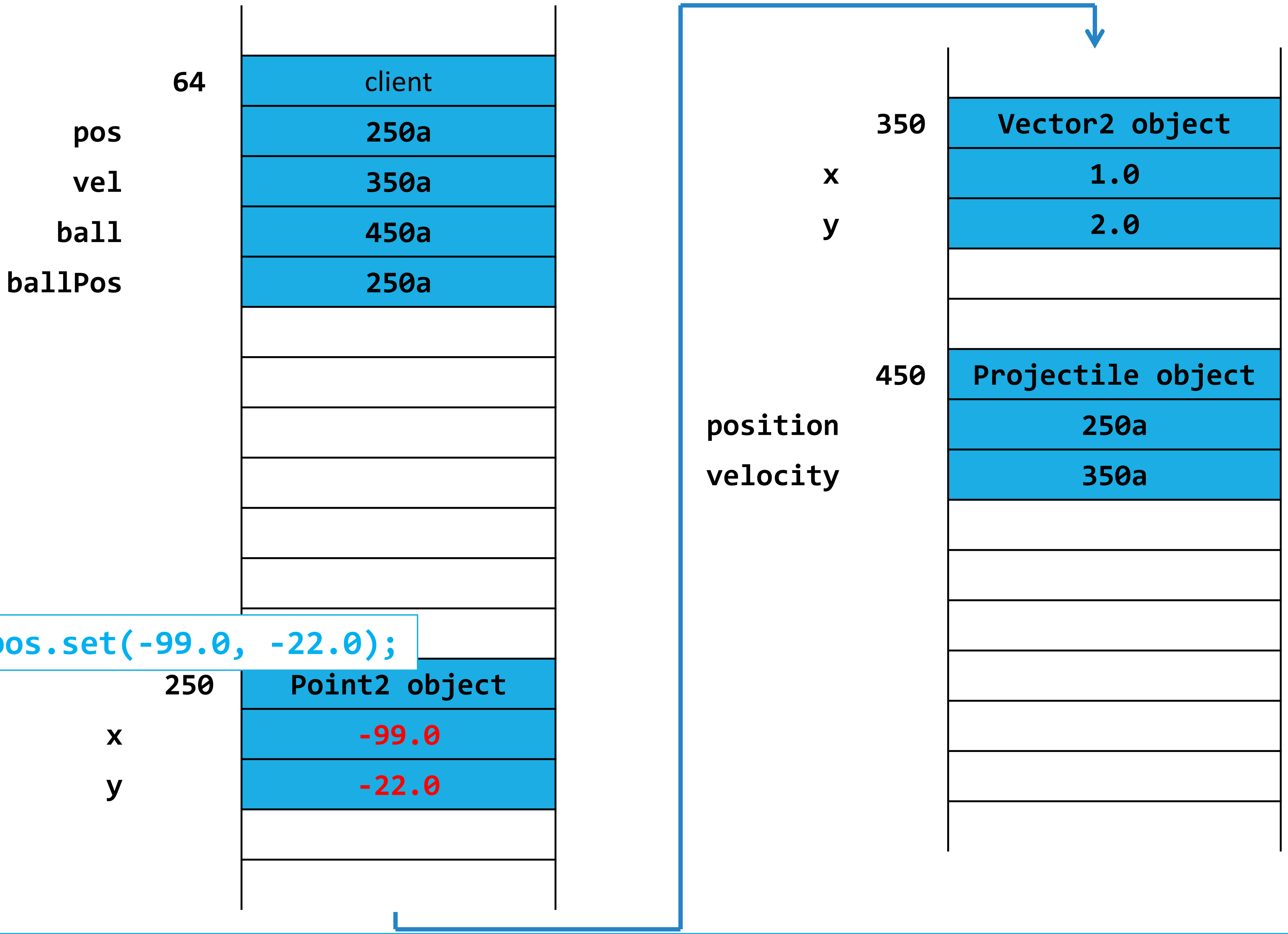
64
client
pos
250a
vel
350a
ball
450a
250
Point2 object
x
10.0
y
20.0
350
Vector2 object
x
1.0
y
2.0
450
Projectile object
position
250a
velocity
350a

```
public static void main(String[] args) {
    Point2 pos = new Point2(10.0, 20.0);
    Vector2 vel = new Vector2(1.0, 2.0);
    Projectile ball = new Projectile();
    ball.setPosition(pos);
    ball.setVelocity(vel);

    // does ball and client share the same objects?
    Point2 ballPos = ball.getPosition();
    System.out.println("same Point2 object?: " + (ballPos == pos));
}
```


64
client
pos
250a
vel
350a
ball
450a
ballPos
250a
ballPos == pos is true
250
Point2 object
x
10.0
y
20.0
350
Vector2 object
x
1.0
y
2.0
450
Projectile object
position
250a
velocity
350a

```
public static void main(String[] args) {
    Point2 pos = new Point2(10.0, 20.0);
    Vector2 vel = new Vector2(1.0, 2.0);
    Projectile ball = new Projectile();
    ball.setPosition(pos);
    ball.setVelocity(vel);

    // does ball and client share the same objects?
    Point2 ballPos = ball.getPosition();
    System.out.println("same Point2 object?: " + (ballPos == pos));

    // client changes pos
    pos.set(-99.0, -22.0);
    System.out.println("ball position: " + ballPos);
}
```

"ball position: (-99.0, -22.0)"



```
pos.set(-99.0, -22.0);
```

`Projectile` as aggregation

if a client gets a reference to the position or velocity of the projectile, then the **client <u>can change</u> these quantities** *without asking the projectile*

Not a flaw of aggregation itself!

- it’s just the consequence of choosing to use aggregation

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 */ );
}
```

$\underbrace{\texttt{new Y(/* suitable arguments */);}}_{\boxed{\text{defensive copy}}}$

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

hopefully there is a copy constructor in Y

```
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*

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: a *privacy leak*

Modify the **`Projectile`** copy constructor so that it uses composition:

```
/**
 * Initialize the projectile so that its position and velocity are
 * equal to the position and velocity of the specified projectile.
 *
 * @param other
 *            a projectile to copy
 */
public Projectile(Projectile other) {
    this.position =
    this.velocity =
}
```

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: a privacy leak

What should **Ball** accessor methods return if they use composition?

```
/**
 * Return the position of the projectile.
 *
 * @return the position of the projectile
 */
public Point2 getPosition() {
    return
}

/**
 * Return the velocity of the projectile.
 *
 * @return the velocity of the projectile
 */
public Vector2 getVelocity() {
    return
}
```

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: a privacy leak

What should **Projectile** mutator methods return if they use composition?

```
/**
 * Set the position of the projectile to the given position.
 *
 * @param position
 *            the new position of the projectile
 */
public void setPosition(Point2 position) {
    this.position =
}

/**
 * Set the velocity of the projectile to the given velocity.
 *
 * @param velocity
 *            the new velocity of the projectile
 */
public void setVelocity(Vector2 velocity) {
    this.velocity =
}
```

Price of Defensive Copying

Defensive copies are required when using composition, but the price of defensive copying is **time** and **memory** needed to create and eventually garbage-collect defensive copies of objects

E.g., the **`BouncingBall`** program calls **`move`** followed by **`getPosition`** every 25 milliseconds

- if **`Projectile`** uses composition then approximately 40 (=1000/25) new **`Point2`** objects are created every second just to move one projectile

High-performance graphics: Bad!

Mission-critical models, one-off calculations… maybe not

Composition

Class Invariants

Class Invariants

- class invariant
  - some property of the state of the object that is established by a constructor and maintained between calls to public methods:
    - **constructor** ensures that the class invariant holds when the constructor is finished running
      - the i. does not necessarily hold **while** the constructor **is** running
    - every **public method** ensures that the class invariant holds when the method is finished running
      - the i. does not necessarily hold **while** the method **is** running

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

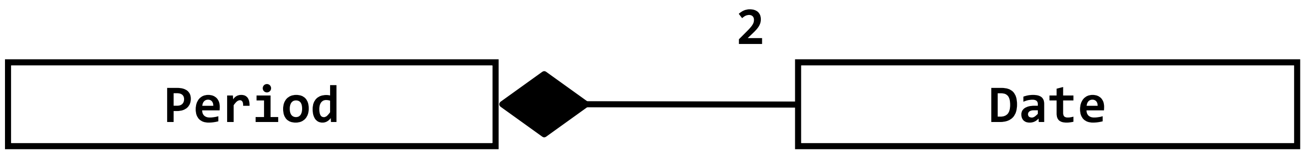
- a period has a start time and an end time
  - end time is always after the start time (this is the class invariant)

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

Period Class



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

java.util.Date

https://docs.oracle.com/javase/8/docs/api/java/util/Date.html

Date has no copy constructor; to copy a Date object one can do the following:

```
Date d = new Date();
Date d2 = new Date(d.getTime());
```

```java
import java.util.Date;


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

Suppose that we implement the **Period** constructor like this:

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

Add one more line of code to show how the client can break the class invariant of **`Period`**:

```
Date start = new Date();
Date end = new Date( start.getTime() + 10000 );
Period p = new Period( start, end );
```

A. `end.setTime( end.getTime() );`

B. `end.setTime( end.getTime() + 10000 );`

C. `start.setTime( end.getTime() + 1 );`

D. `end.setTime( start.getTime() );`

Modify the **Period** constructor so that it uses composition:

```
  /**
   * Initialize the period to the given start and end dates.
   *
   * @param start beginning of the period
   * @param end end of the period; must not precede start
   * @throws IllegalArgumentException if start is after end
   */
   public Period(Date start, Date end) {
       Date startCopy = new Date(start.getTime());
       Date endCopy = new Date(end.getTime());
       if (startCopy.compareTo(endCopy) > 0) {
           throw new IllegalArgumentException("start after end");
       }
       this.start = startCopy;
       this.end = endCopy;
   }
```

Suppose that we implement the **`Period`** copy constructor like this:

```
/**
  * Initialize the period so that it has the same start and end times
  * as the specified period.
  *
  * @param other the period to copy
  */
 public Period(Period other) {
     this.start = other.start;
     this.end = other.end;
 }
```

Modify the **`Period`** copy constructor so that it uses composition:

```
/**
  * Initialize the period so that it has the same start and end times
  * as the specified period.
  *
  * @param other the period to copy
  */
 public Period(Period other) {
     this.start = new Date(other.start.getTime());
     this.end = new Date(other.end.getTime());
 }
```

Suppose that we implement the **`Period`** accessors like this:

```
/**
 * Returns the starting date of the period.
 *
 * @return the starting date of the period
 */
public Date getStart() {
    return this.start;                — privacy leak
}

/**
 * Returns the ending date of the period.
 *
 * @return the ending date of the period
 */
public Date getEnd() {
    return this.end;                  — privacy leak
}
```

Modify the **`Period`** accessors so that they use composition:

```
/**
  * Returns the starting date of the period.
  *
  * @return the starting date of the period
  */
 public Date getStart() {
     return new Date(this.start.getTime());
 }

 /**
  * Returns the ending date of the period.
  *
  * @return the ending date of the period
  */
 public Date getEnd() {
     return new Date(this.end.getTime());
 }
```

Suppose that we implement the **`Period`** mutator like this:

```
/**
  * Sets the starting date of the period.
  *
  * @param newStart the new starting date of the period
  * @return true if the new starting date is earlier than the
  *         current end date; false otherwise
  */
 public boolean setStart(Date newStart) {
     boolean ok = false;
     if (newStart.compareTo(this.end) < 0) {
         this.start = newStart;
         ok = true;
     }
     return ok;
 }
```

Modify the **Period** mutator so that it uses composition:

```
/**
  * Sets the starting date of the period.
  *
  * @param newStart the new starting date of the period
  * @return true if the new starting date is earlier than the
  *         current end date; false otherwise
  */
 public boolean setStart(Date newStart) {
     boolean ok = false;
     Date copy = new Date(newStart.getTime());
     if (copy.compareTo(this.end) < 0) {
         this.start = copy;
         ok = true;
     }
     return ok;
 }
```

Privacy Leaks

a privacy leak occurs when a class exposes a reference to a non-public field (that is not a primitive or immutable)

given a class **X** that is a composition of a **Y**

```
public class X {
  private Y y;
  // …
}
```

these are all examples of privacy leaks

```
public X(Y y) {
  this.y = y;
}
```

```
public X(X other) {
  this.y = other.y;
}
```

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

```
public void setY(Y y) {
  this.y = y;
}
```

Consequences of Privacy Leaks

a privacy leak allows some other object to control the state of the object that leaked the field

- the object state can become inconsistent
  - Example 1: if a **`CreditCard`** exposes a reference to its expiry **`Date`** then a client could set the expiry date to before the issue date
  - Example 2: you obtain a reference to a mutable field and start modifying it nor realizing it modifies the original

Consequences of Privacy Leaks

- a privacy leak allows some other object to control the state of the object that leaked the field
  - it becomes impossible to guarantee class invariants
    - example: if a **`Period`** exposes a reference to one of its **`Date`** objects then the end of the period could be set to before the start of the period

Consequences of Privacy Leaks

a privacy leak allows some other object to control the state of the object that leaked the field

- composition becomes broken because the object no longer *owns* its attribute
  - when an object "dies" its parts may not die with it

Recipe for Immutability

the recipe for immutability in Java is described by Joshua Bloch in the book *Effective Java**

1. Do not provide any methods that can alter the state of the object
2. Prevent the class from being extended
3. Make all fields `final`
4. Make all fields `private`
5. Prevent clients from obtaining a reference to any mutable fields

revisit when we talk about inheritance

*highly recommended reading if you plan on becoming a Java programmer

Composition with an array

- in Java an array is a reference type
  - if a class has a field that is an array then you need to consider whether you should be using aggregation or composition

- an array of elements having primitive type is easier to deal with than an array of elements having reference type
  - if your class has a field that is an array of elements having reference type then things become more complicated

CombinationLock

suppose that you want to implement a simple combination lock

- the lock has a combination having at least 3 digits between 0 and 9
- the lock is always either locked or unlocked

fields:

- an array of int to store the combination
- a boolean to store the locked/unlocked state

```java
import java.util.Arrays;

public class CombinationLock implements Comparable<CombinationLock> {

    /**
     * The combination of this lock.
     */
    private int[] combination;

    /**
     * The state of the lock.
     */
    private boolean isLocked;
```

`CombinationLock`

note that the field **`this.combination`** is not yet initialized

- the constructors must make a new array and store reference to the new array in **`this.combination`**

```java
/**
 * Initializes this lock so that it is locked and it has the
 * combination {@code 9, 9, 9}.
 */
public CombinationLock() {
    this.combination = new int[3];
    for (int i = 0; i < 3; i++) {
        this.combination[i] = 9;
    }
    this.isLocked = true;
}
```

`CombinationLock`

a constructor that receives a combination from the caller must make a new copy of the combination and then validate the copy

- failing to at least make a **copy** of the combination is a potential privacy leak

```
/**
 * Initializes this lock so that it is locked and it has the specified
 * combination. The constructor copies the values from the specified
 * combination into this lock's combination.
 *
 * @param combination
 *            the combination to use for this lock
 * @pre. combination.length greater than or equal to 3
 * @throws IllegalArgumentException
 *             if the number of numbers in the combination is less than 3
 */
public CombinationLock(int[] combination) {
    int[] comboCopy = Arrays.copyOf(combination, combination.length);
    int n = comboCopy.length;
    if (n < 3) {
        throw new IllegalArgumentException();
    }
    this.combination = comboCopy;
    this.isLocked = true;
}
```

`CombinationLock`

- the copy constructor must make a copy of the other lock's combination
  - the easiest way to do this is to use constructor chaining (at the price of performing some unnecessary validation)

```java
/**
 * Initializes this lock by copying the digits from another combination
 * lock into this lock's combination. Also copies the lock state of
 * the other lock (if the other lock is locked then so is this lock,
 * and if other lock is unlocked then so is this lock).
 *
 * @param other the lock to copy
 */
public CombinationLock(CombinationLock other) {
    this(other.combination);
    this.isLocked = other.isLocked;
}
```

`CombinationLock`

not every method needs to perform a defensive copy

- unlocking the lock requires the caller to provide a combination with which to try to unlock the lock with
- the provided combination is not used to modify the lock's combination and the method does not return the lock's combination so there is no privacy leak here

```java
/**
 * Unlocks this lock if the provided combination is equal to the combination
 * of this lock. If the provided combination does not match the combination
 * of this lock then no action is taken.
 *
 * <p>
 * If the specified combination is equal to the combination of this lock then
 * {@code isLocked()} will return {@code false} after this method is called.
 *
 * @param combination
 *            a combination to try to unlock this lock
 */
public void unlock(int[] combination) {
    if (Arrays.equals(this.combination, combination)) {
        this.isLocked = false;
    }
}
```