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

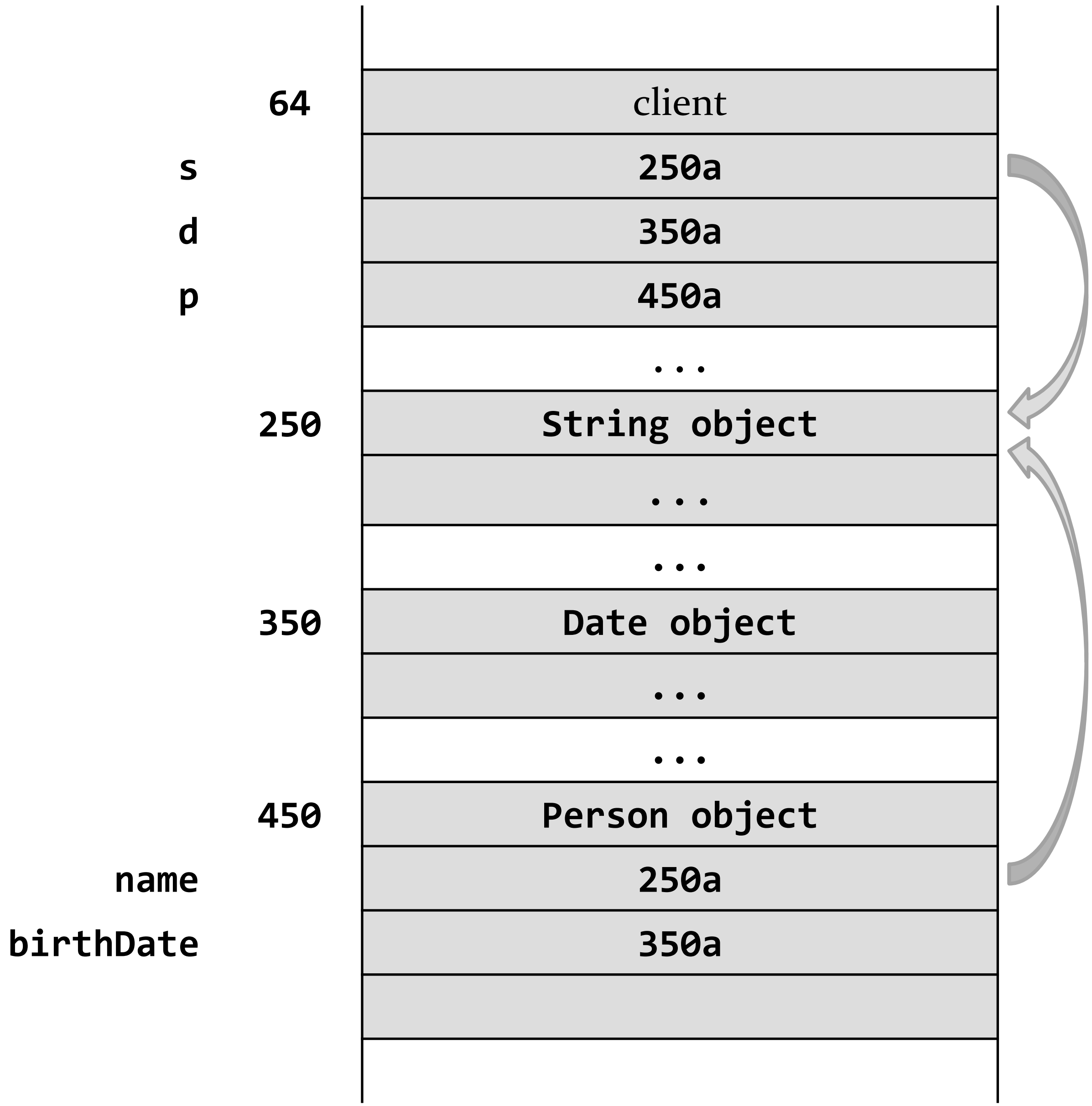
Aggregation

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

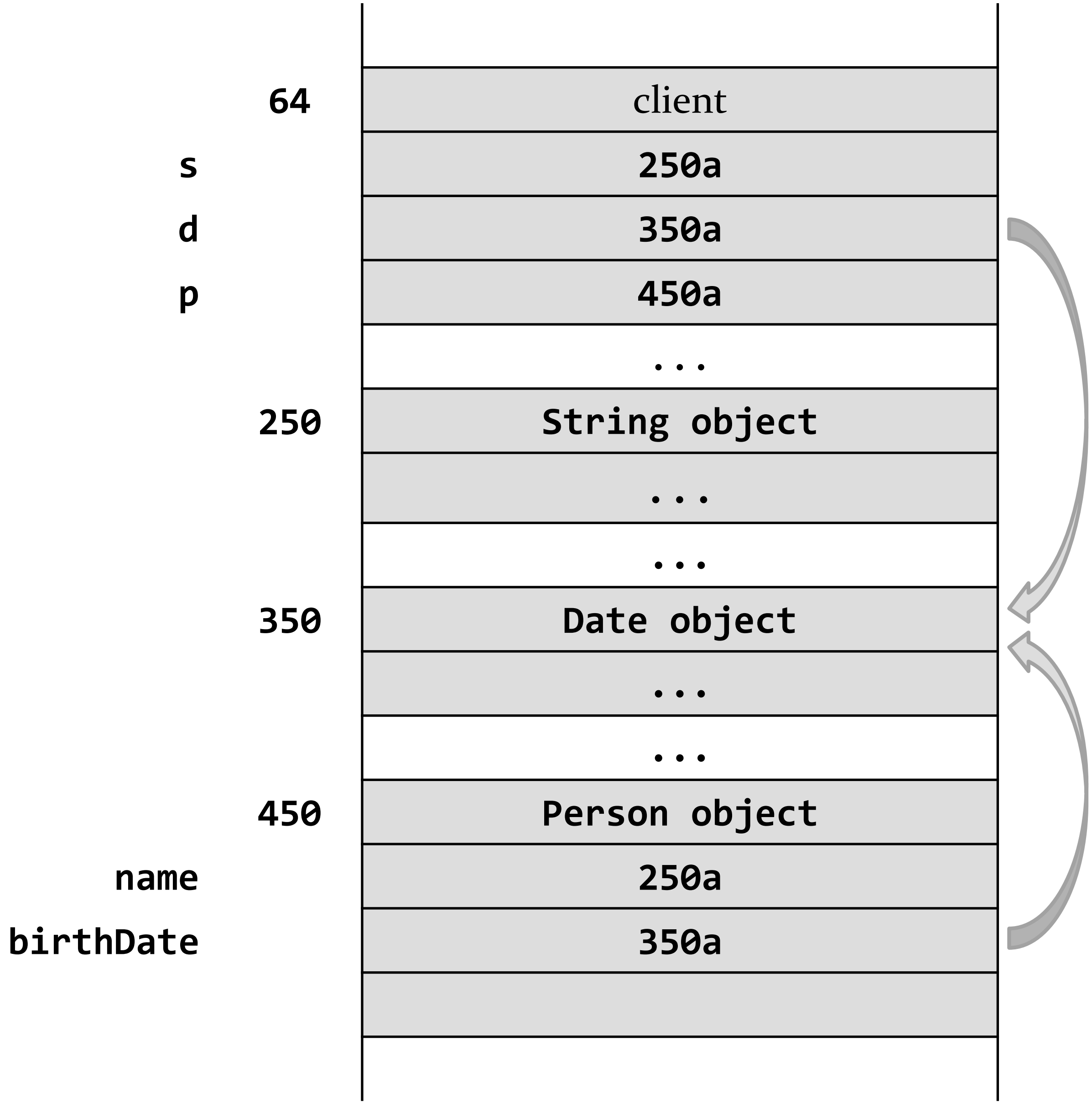
```
public class Person {
  private String name;
  private Date birthDate;

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

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

- the **Person** example uses aggregation
  - notice that the constructor does not make a new 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);
```



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



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

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

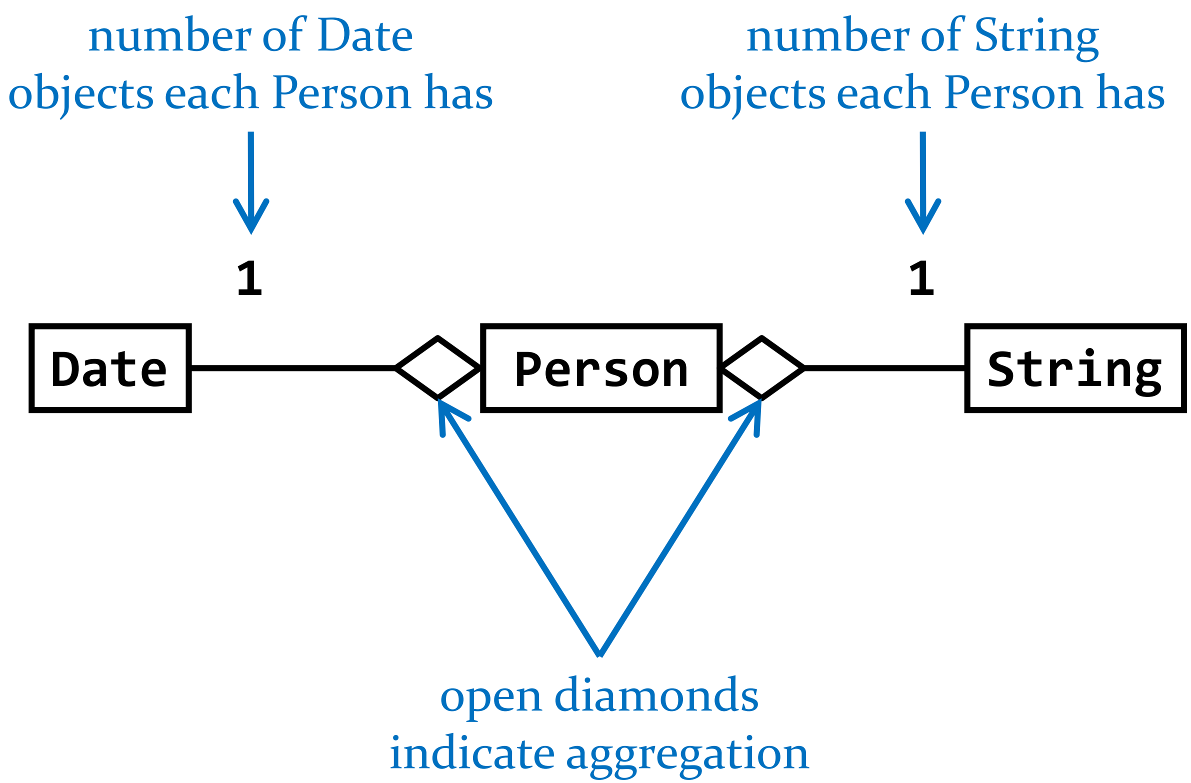
```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(90, 2, 26);  // March 26, 1990
Person p = new Person(s, d);

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

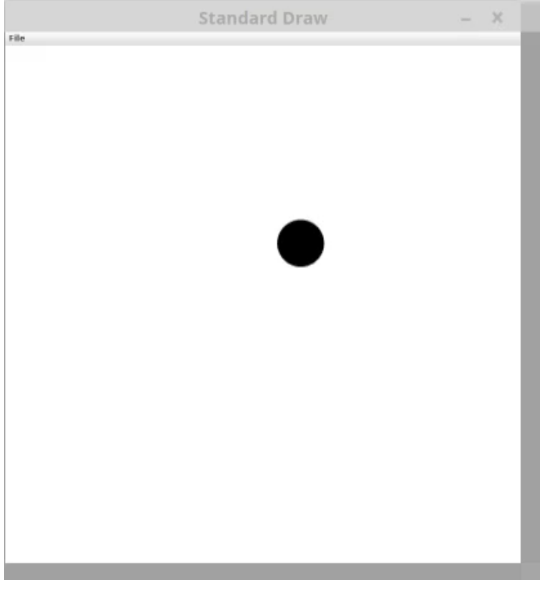


Another Aggregation Example

- consider implementing a bouncing ball whose position is governed by the following equations of motion (see this lab from last year)

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

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



$\mathbf{p}_i$ 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 **`Ball`** has-a **`Point2`** that represents the position of the ball and a **`Vector2`** that represents the velocity of the ball

```java
public class Ball {

    /**
     * The current position of the ball.
     */
    private Point2 position;

    /**
     * The current velocity of the ball.
     */
    private Vector2 velocity;

    /**
     * Gravitational acceleration vector.
     */
    private static final Vector2 G = new Vector2(0.0, -9.81);
```

```java
/**
 * Initialize the ball so that its position and velocity are
 * equal to the given position and velocity.
 *
 * @param position
 *            the position of the ball
 * @param velocity
 *            the velocity of the ball
 */
public Ball(Point2 position, Vector2 velocity) {
    this.position = position;
    this.velocity = velocity;
}
```

```java
/**
 * Return the position of the ball.
 *
 * @return the position of the ball
 */
public Point2 getPosition() {
    return this.position;
}

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

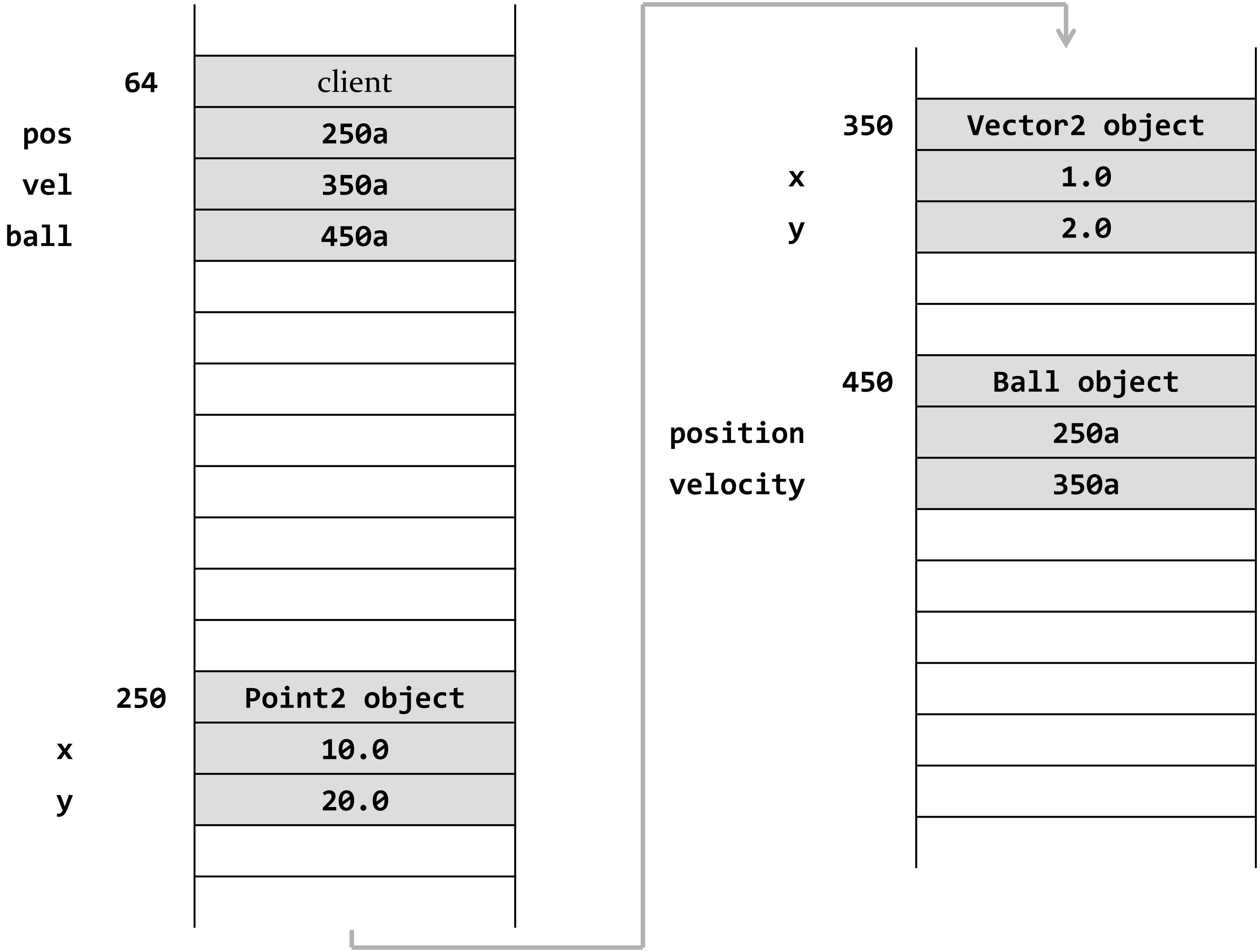
```java
/**
 * Set the position of the ball to the given position.
 *
 * @param position
 *            the new position of the ball
 */
public void setPosition(Point2 position) {
    this.position = position;
}

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

`Ball` as an aggregation

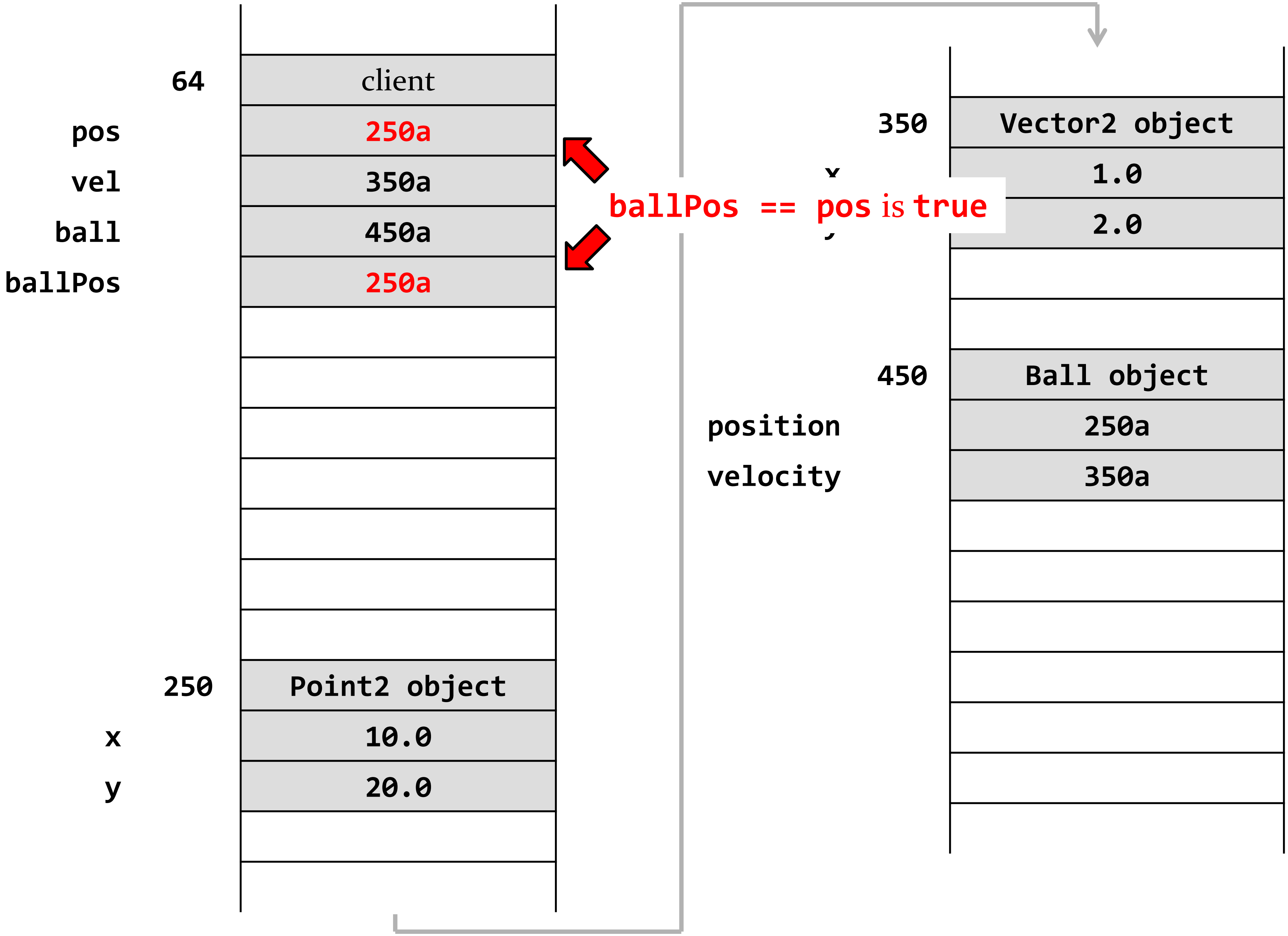
- implementing `Ball` is very easy
- fields
  - are references to existing objects provided by the client
- accessors
  - give clients a reference to the aggregated `Point2` and `Vector2` objects
- mutators
  - set fields to existing object references provided by the client

- we say that the `Ball` fields are *aliases*

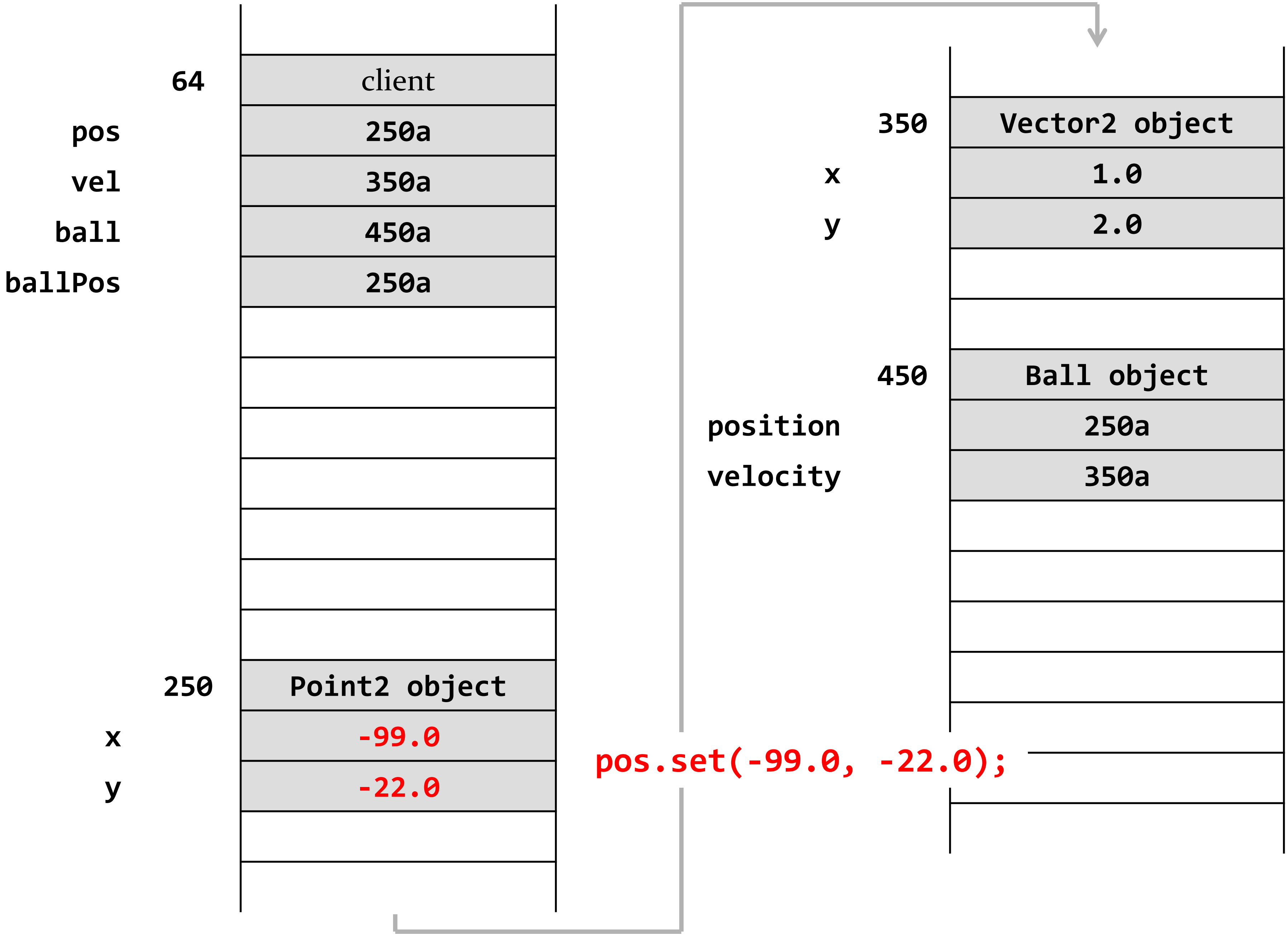
```java
public static void main(String[] args) {
    Point2 pos = new Point2(10.0, 20.0);
    Vector2 vel = new Vector2(1.0, 2.0);
    Ball ball = new Ball(pos, vel);
}
```


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
Ball object
position
250a
velocity
350a

```java
public static void main(String[] args) {
    Point2 pos = new Point2(10.0, 20.0);
    Vector2 vel = new Vector2(1.0, 2.0);
    Ball ball = new Ball(pos, 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
250
Point2 object
x
10.0
y
20.0
ballPos == pos is true
350
Vector2 object
1.0
2.0
450
Ball object
position
250a
velocity
350a

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


64
client
pos
250a
vel
350a
ball
450a
ballPos
250a
250
Point2 object
x
-99.0
y
-22.0
350
Vector2 object
x
1.0
y
2.0
450
Ball object
position
250a
velocity
350a
pos.set(-99.0, -22.0);

`Ball` as aggregation

- if a client gets a reference to the position or velocity of the ball, then the client can change these quantities *without asking the ball*

- this is not a flaw of aggregation
  - 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 */ );
}
```

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

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

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

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

```
/**
 * Initialize the ball so that its position and velocity are
 * equal to the given position and velocity.
 *
 * @param position
 *            the position of the ball
 * @param velocity
 *            the velocity of the ball
 */
public Ball(Point2 position, Vector2 velocity) {
    this.position =
    this.velocity =
}
```

Composition & Copy Constructor

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

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

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

defensive copy

Composition & Copy Constructor

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

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

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

Suppose **`Ball`** had the following copy constructor:

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

What does the following program print?:

```
Point2 p = new Point2();
Vector2 v = new Vector2();
Ball b1 = new Ball(p, v);
Ball b2 = new Ball(b1);
p.setX(-100.0);
b1.setPosition(p);
System.out.println(b2.getPosition());
```

Modify the **Ball** copy constructor so that is uses composition:

```
/**
 * Initialize the ball so that its position and velocity are
 * equal to the position and velocity of the specified ball.
 *
 * @param other
 *           a ball to copy
 */
public Ball(Ball 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

  - this is called a privacy leak

Suppose **Ball** had the following accessor methods:

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

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

What does the following program print?:

```
Ball b = new Ball(new Point2(), new Vector2());
Vector2 v = b.getVelocity();
v.set(-1000.0, 500.0);
System.out.println(b.getVelocity());
```

Modify the **Ball** accessor methods so that they use composition:

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

/**
 * Return the velocity of the ball.
 *
 * @return the velocity of the ball
 */
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);        } defensive copy
  // validate; will throw an exception if copyY is invalid
  this.checkY(copyY);
  this.y = copyY;
}
```

Composition and Mutators

- why is the deep copy required?

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

- if the mutator does this

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

then the client and the **X** object will share the same **Y** object

- this is called a privacy leak

Suppose **`Ball`** had the following mutator methods:

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

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

What does the following program print?:

```
Ball b = new Ball(new Point2(), new Vector2());
Vector2 v = new Vector2(100.0, 200.0);
b.setVelocity(v);
v.set(-1.0, -5.0);
System.out.println(b.getVelocity());
```

Modify the **`Ball`** mutator methods so that they use composition:

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

/**
 * Set the velocity of the ball to the given velocity.
 *
 * @param velocity
 *            the new velocity of the ball
 */
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 garbage collect defensive copies of objects

- recall the **`Ball`** program
  - again, see this lab from last year
  - if you used aggregation then moving the ball could be done without making any defensive copies

```java
/**
 * Moves the ball from its current position using its current
 * velocity accounting for the force of gravity. See the Lab 3
 * document for a description of how to compute the new position
 * and velocity of the ball.
 *
 * @param dt
 *            the time period over which the ball has moved
 * @return the new position of the ball
 */
public Point2 move(double dt) {
    Vector2 dp1 = Lab3Util.multiply(dt, this.velocity);
    Vector2 dp2 = Lab3Util.multiply(0.5 * dt * dt, Ball.G);
    Vector2 dp = Lab3Util.add(dp1, dp2);
    this.position = Lab3Util.add(this.position, dp);
    Vector2 dv = Lab3Util.multiply(dt, Ball.G);
    this.velocity.add(dv);
    return this.position;
}
```

Price of Defensive Copying

- if we use composition to implement **`Ball`** then move must return a defensive copy of **`this.position`**
- this doesn't seem like such a big deal until you realize that the **`BouncingBall`** program causes the ball to move many times each second

Composition (Part 2)

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
  - in other words:
    - the constructor ensures that the class invariant holds when the constructor is finished running
      - the invariant 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 invariant does not necessarily hold while the method is running

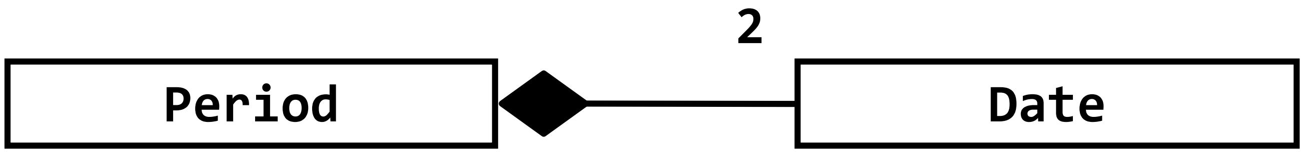
Period Class

- 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

```java
import java.util.Date;

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

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

```
/**
 * 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 );
```

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) {
    if (start.compareTo(end) > 0) {
        throw new IllegalArgumentException("start after end");
    }
    this.start =
    this.end =
}
```

Suppose that we implement the **Period** copy constructor like so:

```java
/**
 * 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;
}
```

What does the following code fragment print?:

```
Date start = new Date();
Date end = new Date( start.getTime() + 10000 );
Period p1 = new Period( start, end );
Period p2 = new Period( p1 );
System.out.println( p1.getStart() == p2.getStart() );
System.out.println( p1.getEnd() == p2.getEnd() );
```

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 =
       this.end =
   }
```

Suppose that we implement the **Period** accessors like so:

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

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

Add one more line of code that uses an accessor method 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 );
```

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
}

/**
 * Returns the ending date of the period.
 *
 * @return the ending date of the period
 */
public Date getEnd() {
    return
}
```

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

```
/**
 * 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;
}
```

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 );
p.setStart( start );
```

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;

    if (          .compareTo(this.end) < 0) {
        this.start =
        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: if a **`CreditCard`** exposes a reference to its expiry **`Date`** then a client could set the expiry date to before the issue date

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

Immutability and Composition

- why is Item 5 of the Recipe for Immutability needed?