

Composition (Part 2)

Price of Defensive Copying

- ▶ defensive copies are often required, but the price of defensive copying is time and memory needed to create and garbage collect lots of objects

- ▶ run triangle demo program

Period Class

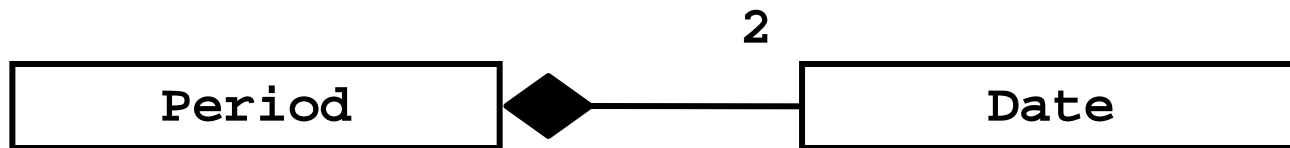
- ▶ adapted from Effective Java by Joshua Bloch
 - ▶ available online at <http://www.informit.com/articles/article.aspx?p=31551&seqNum=2>
- ▶ we want to implement a class that represents a period of time
 - ▶ a period has a start time and an end time
 - ▶ end time is always after the start time

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 composition
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 = start;
        this.end = end;
    }
}
```

Test Your Knowledge

1. Is **Date** mutable or immutable?
2. Is **Period** implementing aggregation or composition?
3. Add 1 more line of client code to the following that shows how the client can break the class invariant:

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

4. Fix the constructor.

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

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


Test Your Knowledge

1. Add 1 more line of client code to the following that shows how the client can break the class invariant using either of the **start** or **end** methods

```
Date start = new Date();
```

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

```
Period p = new Period( start, end );
```

```
/**
 * Creates a time period by copying another time period.
 * @param other the time period to copy
 */
public Period( Period other )
{
    this.start = other.start;
    this.end = other.end;
}
```

Test Your Knowledge

1. What does the following program 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() );
```

2. Fix the copy constructor.

Date does not provide a copy constructor. To copy a **Date** object **d**:

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

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

Test Your Knowledge

1. Add 1 more line of client code to the following that shows how the client can break the class invariant

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

2. Fix the accessors and **setStart**.

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

revisit when we talk about inheritance

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 composition

Immutability and Composition

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

Collections as Attributes

Still Aggregation and Composition

Motivation

- ▶ 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
 - ▶ from the notes, a student has-a collection of GPAs and has-a collection of courses
 - ▶ a polygonal model has-a collection of triangles*

*polygons, actually, but triangles are easier to work with

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

dates

d1

d2

d3

200

client invocation

200

500

600

700

...

ArrayList object

500

600

700



Test Your Knowledge

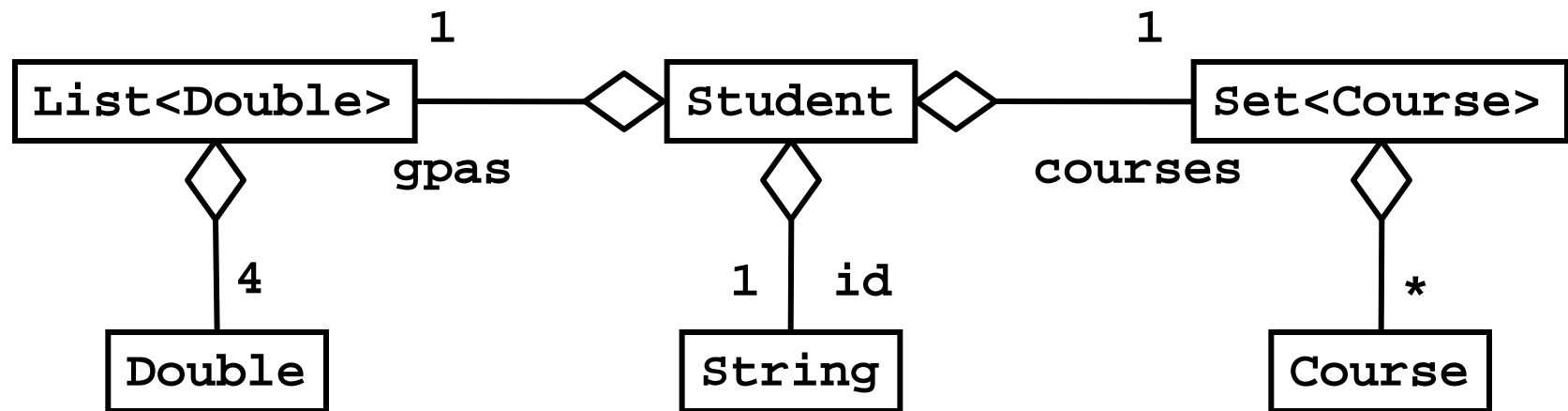
1. What does the following print?

```
ArrayList<Point> pts = new ArrayList<Point>();  
Point p = new Point(0., 0., 0.);  
pts.add(p);  
p.setX( 10.0 );  
System.out.println(p);  
System.out.println(pts.get(0));
```

2. Is an **ArrayList<X>** an aggregation of **X** or a composition of **X**?

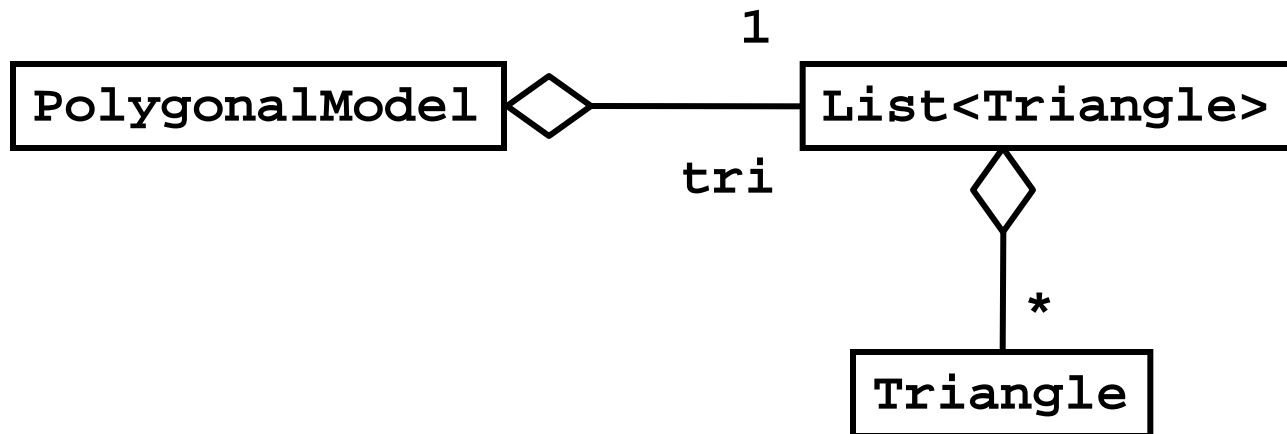
Student Class (from notes)

- ▶ 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



Iterable Interface

- ▶ implementing this interface allows an object to be the target of the "foreach" statement
- ▶ must provide the following method

```
Iterator<T> iterator()
```

Returns an iterator over a set of elements of type T.

PolygonalModel

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

    public PolygonalModel()
    {
        this.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
 - ▶ [notes 4.3.3] 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

Test Your Knowledge

1. Suppose you have a `PolygonalModel p1` that has 100 `Triangles`. What does the following code print?

```
PolygonalModel p2 = new PolygonalModel(p1);  
p2.clear();  
System.out.println( p2.size() );  
System.out.println( p1.size() );
```



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> dCopy = new ArrayList<Date>();
for(Date d : dates)
{
    dCopy.add(new Date(d.getTime()));
}
```

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

constructor invocation
creates a new object