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

Triangle Aggregation

- if a client gets a reference to one of the triangle's points, then the client can change the position of the point *without asking the triangle*

```
pointB = new Point(0.0, 1.0, -3.0);
tri = new Triangle(new Point(-1.0, -1.0, -3.0),
                   pointB,
                   new Point(2.0, 0.0, -3.0));

// 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();

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

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

draw the triangle by asking **tri** for the coordinates of each of its points

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

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

Period Class

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

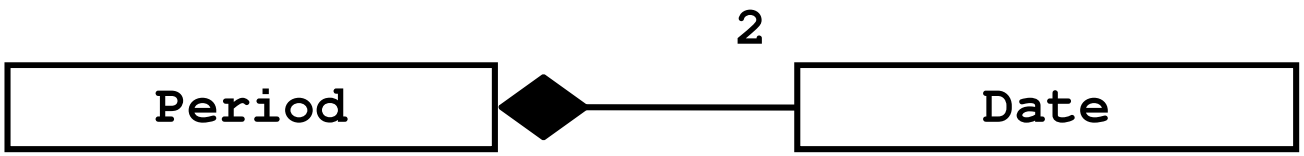
- we want to implement a class that represents a period of time
  - a period has a start time and an end time
    - end time is always after the start time

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

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

Immutability and Composition

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

Collections as Attributes

Still Aggregation and Composition

Motivation

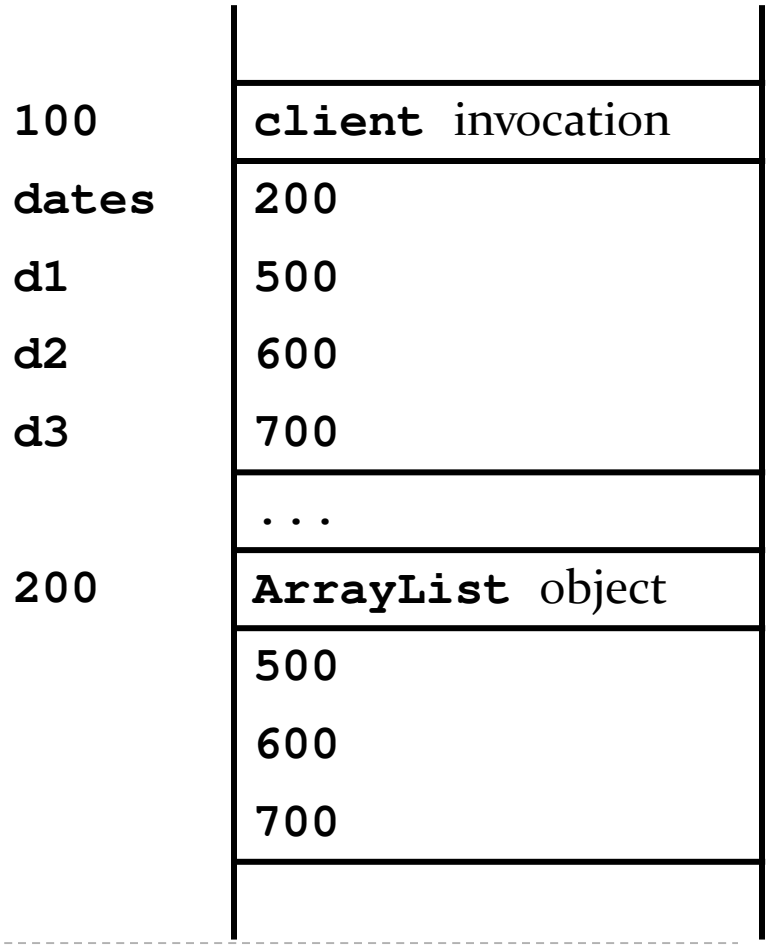
- often you will want to implement a class that has-a collection as an attribute
  - a university has-a collection of faculties and each faculty has-a collection of schools and departments
  - a molecule has-a collection of atoms
  - a person has-a collection of acquaintances
  - from the notes, a student 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);
```



Test Your Knowledge

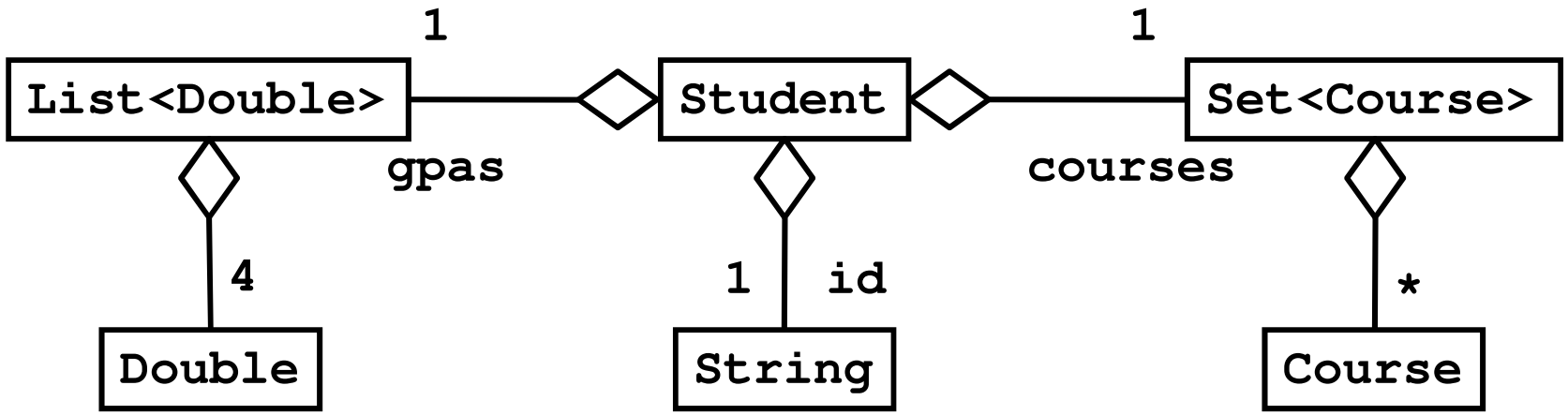
1. What does the following print?

```
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`**?

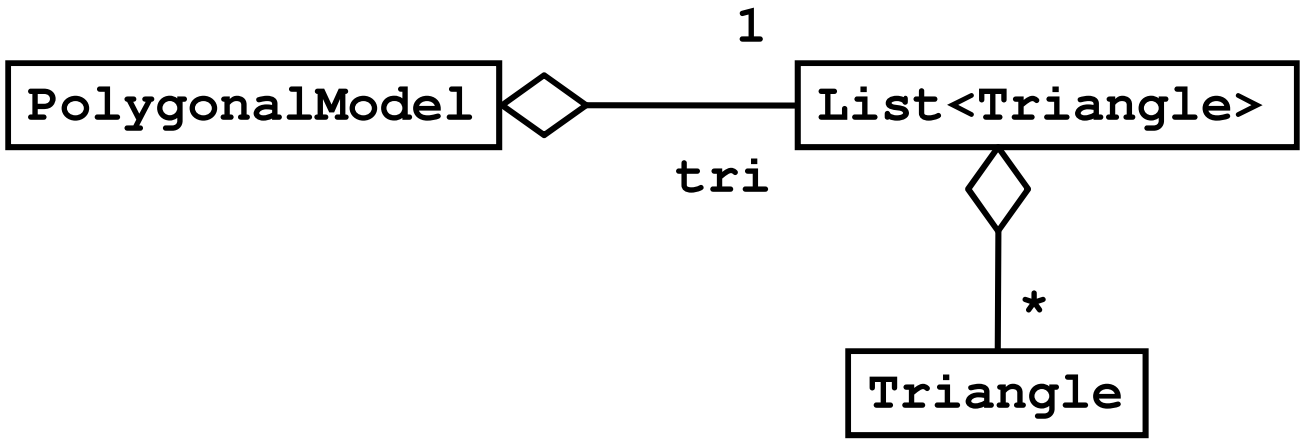
Student Class (from notes)

- 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 `X`s

- 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 **`Triangle`**s. 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);
}
```

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

`add` does not create new objects

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

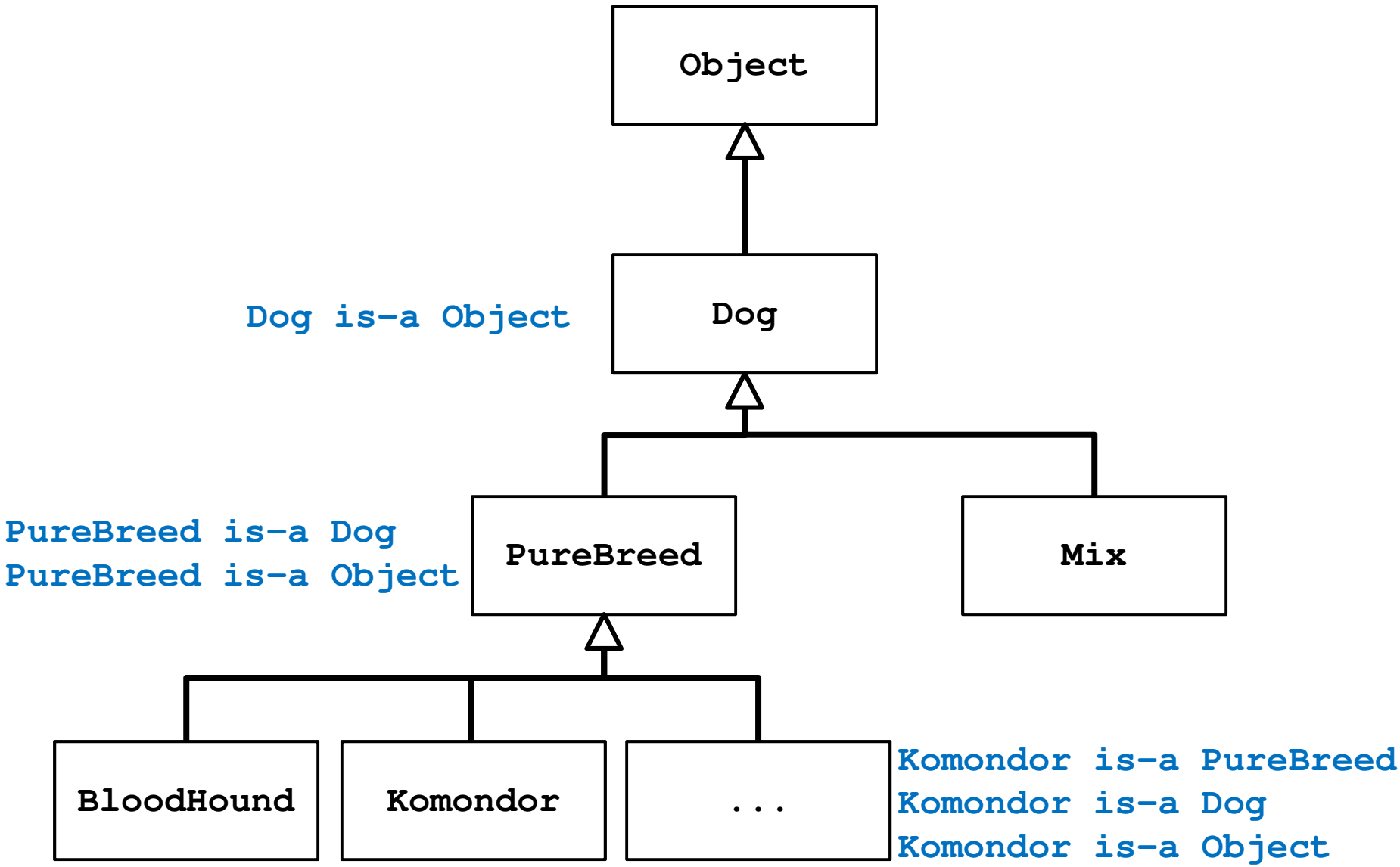
constructor invocation creates a new object

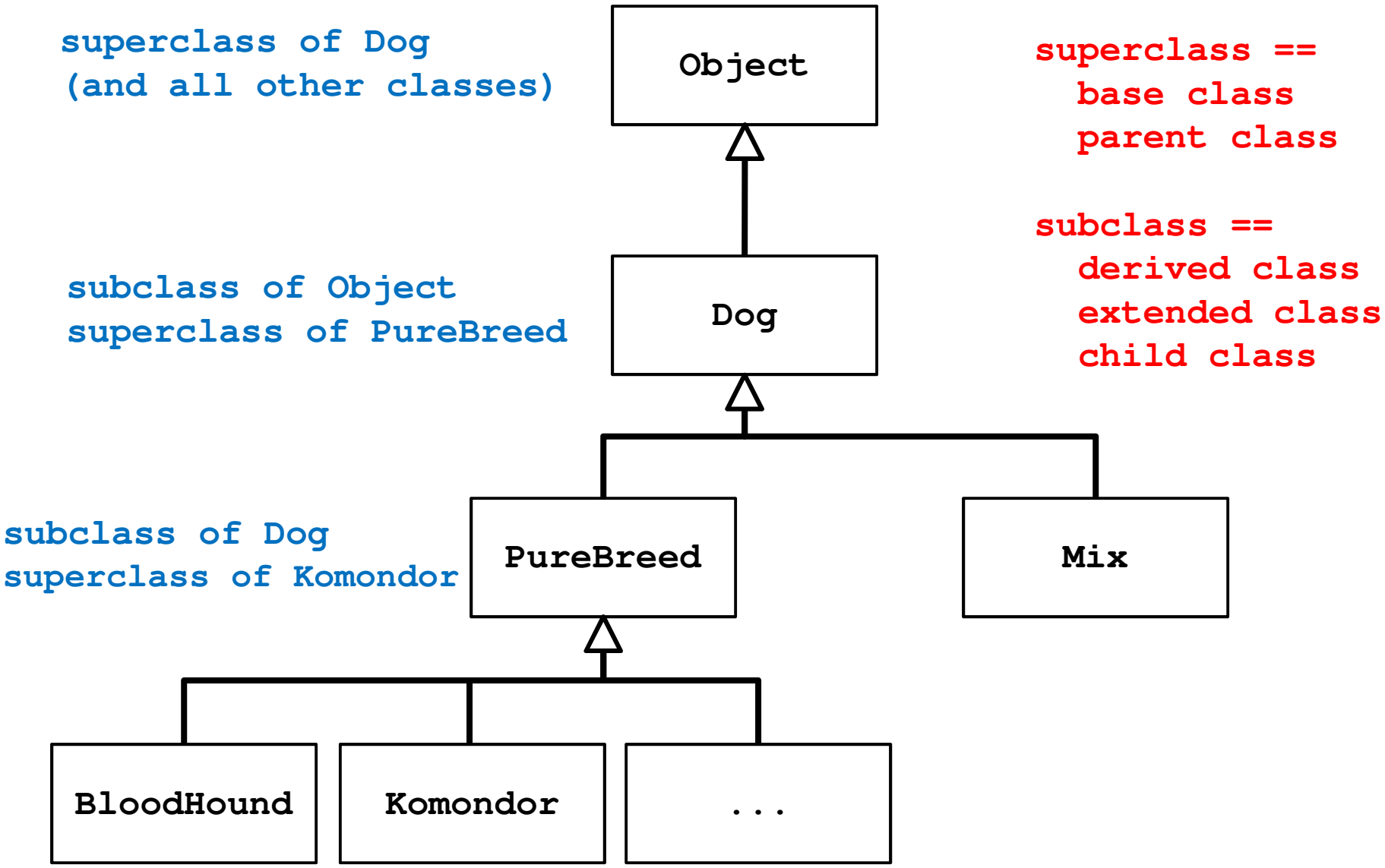
Inheritance

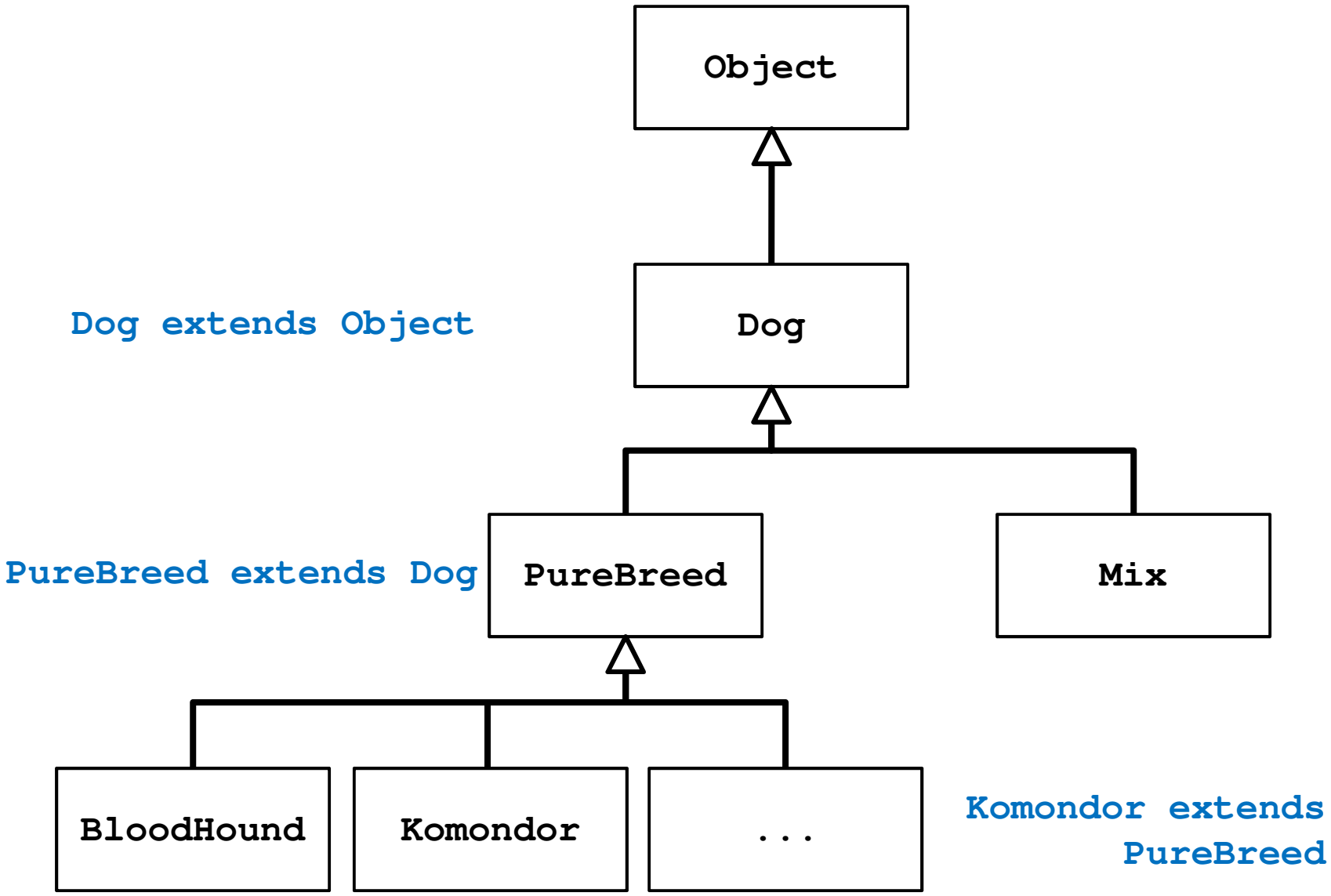
Notes Chapter 6

Inheritance

- you know a lot about an object by knowing its class
  - for example what is a Komondor?


Object
Dog
Dog is-a Object
PureBreed
PureBreed is-a Dog
PureBreed is-a Object
Mix
BloodHound
Komondor
...
Komondor is-a PureBreed
Komondor is-a Dog
Komondor is-a Object


superclass of Dog
(and all other classes)
Object
superclass ==
base class
parent class
subclass ==
derived class
extended class
child class
subclass of Object
superclass of PureBreed
Dog
subclass of Dog
superclass of Komondor
PureBreed
Mix
BloodHound
Komondor
...


Object
Dog extends Object
Dog
PureBreed extends Dog
PureBreed
Mix
BloodHound
Komondor
...
Komondor extends PureBreed

Some Definitions

- we say that a subclass is derived from its superclass
- with the exception of **`Object`**, every class in Java has one and only one superclass
  - Java only supports *single inheritance*
- a class **`X`** can be derived from a class that is derived from a class, and so on, all the way back to **`Object`**
  - **`X`** is said to be descended from all of the classes in the inheritance chain going back to **`Object`**
  - all of the classes **`X`** is derived from are called ancestors of **`X`**

Why Inheritance?

- a subclass inherits all of the non-private members (attributes and methods ***but not constructors***) from its superclass
  - if there is an existing class that provides some of the functionality you need you can derive a new class from the existing class
  - the new class has direct access to the **`public`** and **`protected`** attributes and methods without having to re-declare or re-implement them
  - the new class can introduce new fields and methods
  - the new class can re-define (override) its superclass methods

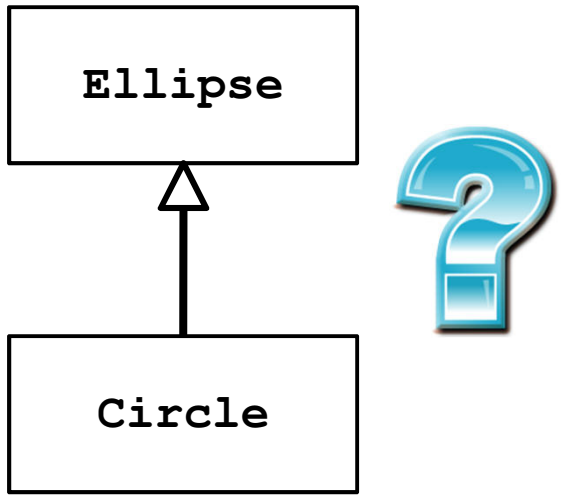
Is-A

- inheritance models the is-a relationship between classes
- from a Java point of view, is-a means you can use a derived class instance in place of an ancestor class instance

```
public someMethod(Dog dog)
{ // does something with dog }

// client code of someMethod

Komondor shaggy = new Komondor();
someMethod( shaggy );

Mix mutt = new Mix ();
someMethod( mutt );
```

Is-A Pitfalls

- is-a has nothing to do with the real world
- is-a has everything to do with how the implementer has modelled the inheritance hierarchy
- the classic example:
  - **`Circle`** is-a **`Ellipse`**?



`Circle` is-a `Ellipse`?

- if `Ellipse` can do something that `Circle` cannot, then `Circle` is-a `Ellipse` is false
  - remember: is-a means you can substitute a derived class instance for one of its ancestor instances
    - if `Circle` cannot do something that `Ellipse` can do then you cannot (safely) substitute a `Circle` instance for an `Ellipse` instance

```
// method in Ellipse
/*
 * Change the width and height of the ellipse.
 * @param width The desired width.
 * @param height The desired height.
 * @pre. width > 0 && height > 0
 */
public void setSize(double width, double height)
{
  this.width = width;
  this.height = height;
}
```

- there is no good way for **`Circle`** to support **`setSize`** (assuming that the attributes **`width`** and **`height`** are always the same for a **`Circle`**) because clients expect **`setSize`** to set both the width and height
- can't **`Circle`** override **`setSize`** so that it throws an exception if **`width != height`**?
  - no; this will surprise clients because **`Ellipse`** **`setSize`** does not throw an exception if **`width != height`**
- can't **`Circle`** override **`setSize`** so that it sets **`width == height`**?
  - no; this will surprise clients because **`Ellipse`** **`setSize`** says that the **`width`** and **`height`** can be different

- But I have a Ph.D. in Mathematics, and I'm *sure* a Circle is a kind of an Ellipse! Does this mean Marshall Cline is stupid? Or that C++ is stupid? Or that OO is stupid? [C++ FAQs http://www.parashift.com/c++-faq-lite/proper-inheritance.html#faq-21.8]
  - Actually, it doesn't mean any of these things. But I'll tell you what it does mean — you may not like what I'm about to say: it means your intuitive notion of "kind of" is leading you to make bad inheritance decisions. Your tummy is lying to you about what good inheritance really means — stop believing those lies.

- what if there is no **`setSize`** method?
  - if a **`Circle`** can do everything an **`Ellipse`** can do then **`Circle`** can extend **`Ellipse`**

Implementing Inheritance

- suppose you want to implement an inheritance hierarchy that represents breeds of dogs for the purpose of helping people decide what kind of dog would be appropriate for them
- many possible fields:
  - appearance, size, energy, grooming requirements, amount of exercise needed, protectiveness, compatibility with children, etc.
  - we will assume two fields measured on a 10 point scale
    - size from 1 (small) to 10 (giant)
    - energy from 1 (lazy) to 10 (high energy)

Dog

```
public class Dog extends Object
{
  private int size;
  private int energy;

  // creates an "average" dog
  Dog()
  {  this(5, 5); }

  Dog(int size, int energy)
  {  this.setSize(size);  this.setEnergy(energy);  }
```

```
  public int getSize()
  { return this.size; }

  public int getEnergy()
  { return this.energy; }

  public final void setSize(int size)
  { this.size = size; }

  public final void setEnergy(int energy)
  { this.energy = energy; }
}
```

why final? stay tuned...

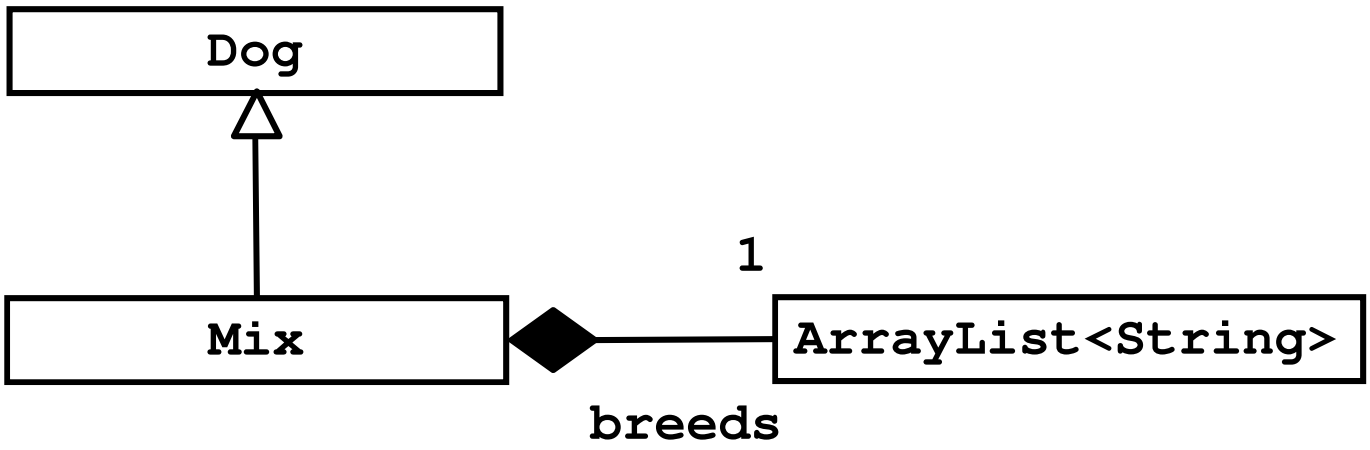
What is a Subclass?

- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
- inheritance does more than copy the API of the superclass
  - the derived class contains a subobject of the parent class
  - the superclass subobject needs to be constructed (just like a regular object)
    - the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

Constructors of Subclasses

1. the first line in the body of every constructor ***must*** be a call to another constructor
   - if it is not then Java will insert a call to the superclass default constructor
     - if the superclass default constructor does not exist or is private then a compilation error occurs
2. a call to another constructor can only occur on the first line in the body of a constructor
3. the superclass constructor must be called during construction of the derived class

Mix UML Diagram



Mix (version 1)

```
public final class Mix extends Dog
{ // no declaration of size or energy; inherited from Dog
  private ArrayList<String> breeds;

  public Mix ()
  { // call to a Dog constructor
    super();
    this.breeds = new ArrayList<String>();
  }

  public Mix(int size, int energy)
  { // call to a Dog constructor
    super(size, energy);
    this.breeds = new ArrayList<String>();
  }
```

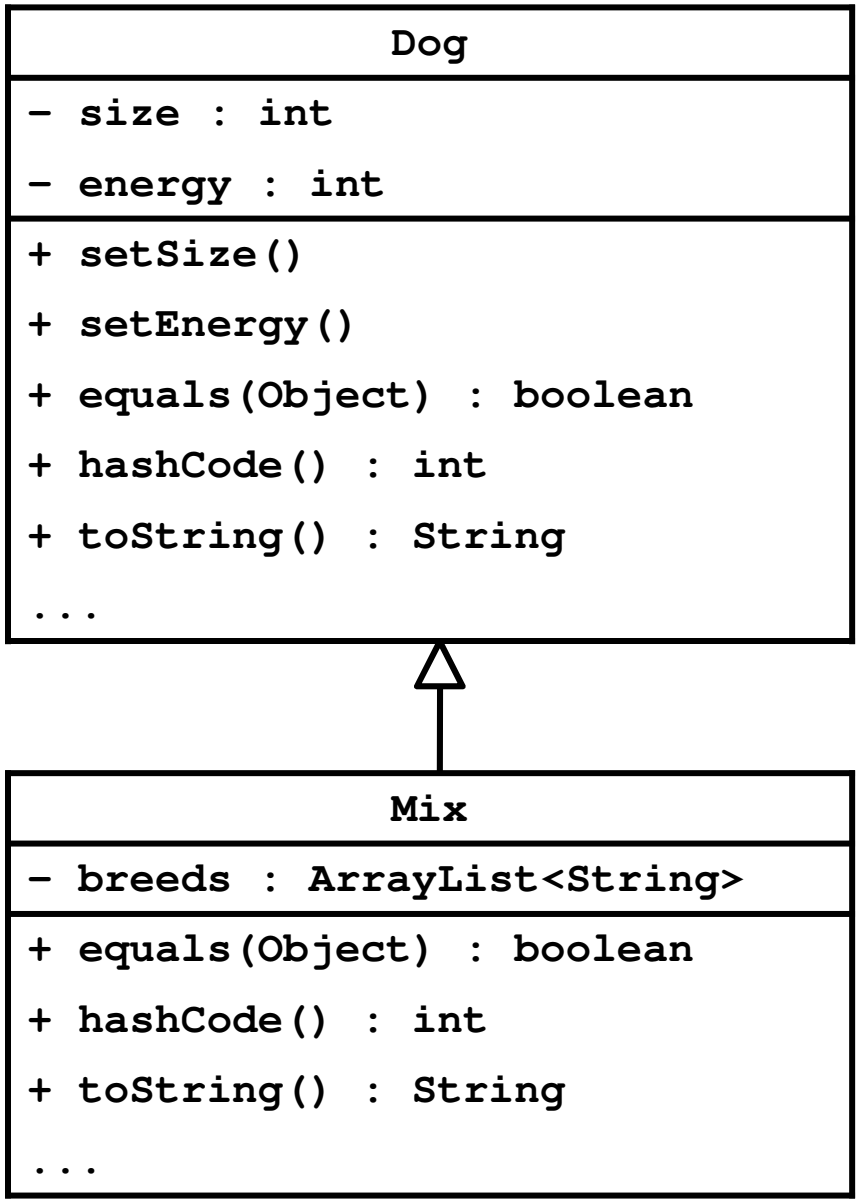
```
public Mix(int size, int energy,
           ArrayList<String> breeds)
{ // call to a Dog constructor
  super(size, energy);
  this.breeds = new ArrayList<String>(breeds);
}
```

Mix (version 2)

```
public final class Mix extends Dog
{ // no declaration of size or energy; inherited from Dog
  private ArrayList<String> breeds;

  public Mix ()
  { // call to a Mix constructor
    this(5, 5);
  }

  public Mix(int size, int energy)
  { // call to a Mix constructor
    this(size, energy, new ArrayList<String>());
  }
```

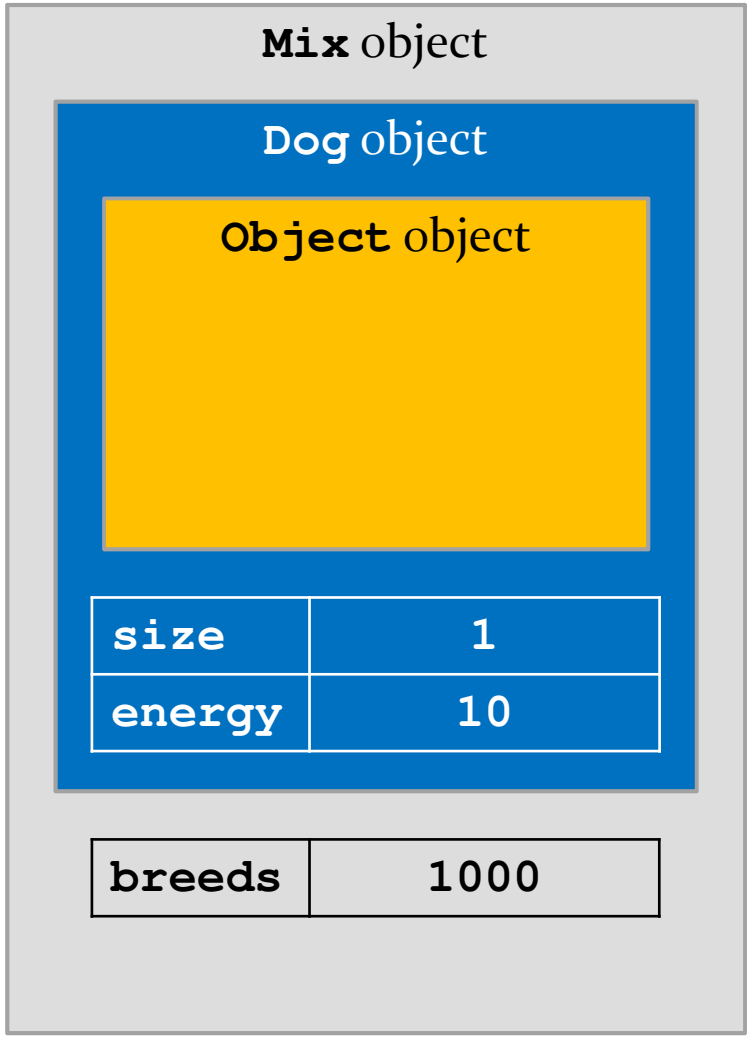
```
public Mix(int size, int energy,
           ArrayList<String> breeds)
{ // call to a Dog constructor
  super(size, energy);
  this.breeds = new ArrayList<String>(breeds);
}
```

- why is the constructor call to the superclass needed?
  - because **`Mix`** is-a **`Dog`** and the **`Dog`** part of **`Mix`** needs to be constructed


Dog
- size : int
- energy : int
+ setSize()
+ setEnergy()
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
...
Mix
- breeds : ArrayList<String>
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
...

```
Mix mutt = new Mix(1, 10);
```

1. **Mix** constructor starts running
- creates new **Dog** subobject by invoking the **Dog** constructor
    2. **Dog** constructor starts running
    - creates new **Object** subobject by (silently) invoking the **Object** constructor
        3. **Object** constructor runs
    - sets **size** and **energy**
- creates a new empty **ArrayList** and assigns it to **breeds**



Invoking the Superclass Ctor

- why is the constructor call to the superclass needed?
  - because **`Mix`** is-a **`Dog`** and the **`Dog`** part of **`Mix`** needs to be constructed
    - similarly, the **`Object`** part of **`Dog`** needs to be constructed

Invoking the Superclass Ctor

- a derived class can only call its own constructors or the constructors of its immediate superclass
  - **`Mix`** can call **`Mix`** constructors or **`Dog`** constructors
  - **`Mix`** cannot call the **`Object`** constructor
    - **`Object`** is not the immediate superclass of **`Mix`**
  - **`Mix`** cannot call **`PureBreed`** constructors
    - cannot call constructors across the inheritance hierarchy
  - **`PureBreed`** cannot call **`Komondor`** constructors
    - cannot call subclass constructors

Constructors & Overridable Methods

- if a class is intended to be extended then its constructor must not call an overridable method
  - Java does not enforce this guideline
- why?
  - recall that a derived class object has inside of it an object of the superclass
  - the superclass object is always constructed first, then the subclass constructor completes construction of the subclass object
  - the superclass constructor will call the overridden version of the method (the subclass version) even though the subclass object has not yet been constructed

Superclass Ctor & Overridable Method

```
public class SuperDuper
{
  public SuperDuper()
  {
    // call to an over-ridable method; bad
    this.overrideMe();
  }

  public void overrideMe()
  {
    System.out.println("SuperDuper overrideMe");
  }
}
```

Subclass Overrides Method

```
public class SubbyDubby extends SuperDuper {
  private final Date date;

  public SubbyDubby()
  {  super();  this.date = new Date();  }

  @Override public void overrideMe()
  {  System.out.print("SubbyDubby overrideMe : ");
     System.out.println( this.date );  }

  public static void main(String[] args)
  {  SubbyDubby sub = new SubbyDubby();
     sub.overrideMe();                    }
}
```

- the programmer's intent was probably to have the program print:

```
SuperDuper overrideMe
SubbyDubby overrideMe : <the date>
```

or, if the call to the overridden method was intentional

```
SubbyDubby overrideMe : <the date>
SubbyDubby overrideMe : <the date>
```

- but the program prints:

```
SubbyDubby overrideMe : null
SubbyDubby overrideMe : <the date>
```

final attribute in two different states!

What's Going On?

1. **`new SubbyDubby()`** calls the **`SubbyDubby`** constructor
2. the **`SubbyDubby`** constructor calls the **`SuperDuper`** constructor
3. the **`SuperDuper`** constructor calls the method **`overrideMe`** which is overridden by **`SubbyDubby`**
4. the **`SubbyDubby`** version of **`overrideMe`** prints the **`SubbyDubby date`** attribute which has not yet been assigned to by the **`SubbyDubby`** constructor (so **`date`** is null)
5. the **`SubbyDubby`** constructor assigns **`date`**
6. **`SubbyDubby overrideMe`** is called by the client

- remember to make sure that your base class constructors only call **`final`** methods or **`private`** methods
  - if a base class constructor calls an overridden method, the method will run in an unconstructed derived class

Other Methods

- methods in a subclass will often need or want to call methods in the immediate superclass
  - a new method in the subclass can call any **`public`** or **`protected`** method in the superclass without using any special syntax
- a subclass can override a **`public`** or **`protected`** method in the superclass by declaring a method that has the same signature as the one in the superclass
  - a subclass method that overrides a superclass method can call the overridden superclass method using the **`super`** keyword

Dog equals

- we will assume that two **Dog**s are equal if their size and energy are the same

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if(obj != null && this.getClass() == obj.getClass())
  {
    Dog other = (Dog) obj;
    eq = this.getSize() == other.getSize() &&
         this.getEnergy() == other.getEnergy();
  }
  return eq;
}
```

Mix equals (version 1)

- two Mix instances are equal if their Dog subobjects are equal and they have the same breeds

```
@Override public boolean equals(Object obj)
{ // the hard way
  boolean eq = false;
  if(obj != null && this.getClass() == obj.getClass()) {
    Mix other = (Mix) obj;
    eq = this.getSize() == other.getSize() &&
         this.getEnergy() == other.getEnergy() &&
         this.breeds.size() == other.breeds.size() &&
         this.breeds.containsAll(other.breeds);
  }
  return eq;
}
```

subclass can call public method of the superclass

Mix equals (version 2)

- two Mix instances are equal if their Dog subobjects are equal and they have the same breeds
  - Dog equals already tests if two Dog instances are equal
  - Mix equals can call Dog equals to test if the Dog subobjects are equal, and then test if the breeds are equal
- also notice that Dog equals already checks that the Object argument is not null and that the classes are the same
  - Mix equals does not have to do these checks again

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if(super.equals(obj))
  { // the Dog subobjects are equal
    Mix other = (Mix) obj;
    eq = this.breeds.size() == other.breeds.size() &&
         this.breeds.containsAll(other.breeds);
  }
  return eq;
}
```

subclass method that overrides a superclass method can call the overridden superclass method

Dog toString

```
@Override public String toString()
{
  String s = "size " + this.getSize() +
             "energy " + this.getEnergy();
  return s;
}
```

Mix toString

```
@Override public String toString()
{
  StringBuffer b = new StringBuffer();
  b.append(super.toString());
  for(String s : this.breeds)
    b.append(" " + s);
  b.append(" mix");
  return b.toString();
}
```

Dog hashCode

```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = 1;
  result = prime * result + this.getEnergy();
  result = prime * result + this.getSize();
  return result;
}
```

Mix hashCode

```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + this.breeds.hashCode();
  return result;
}
```

Mix Memory Diagram

- inherited from superclass
- private in superclass
- not accessible by name to Mix