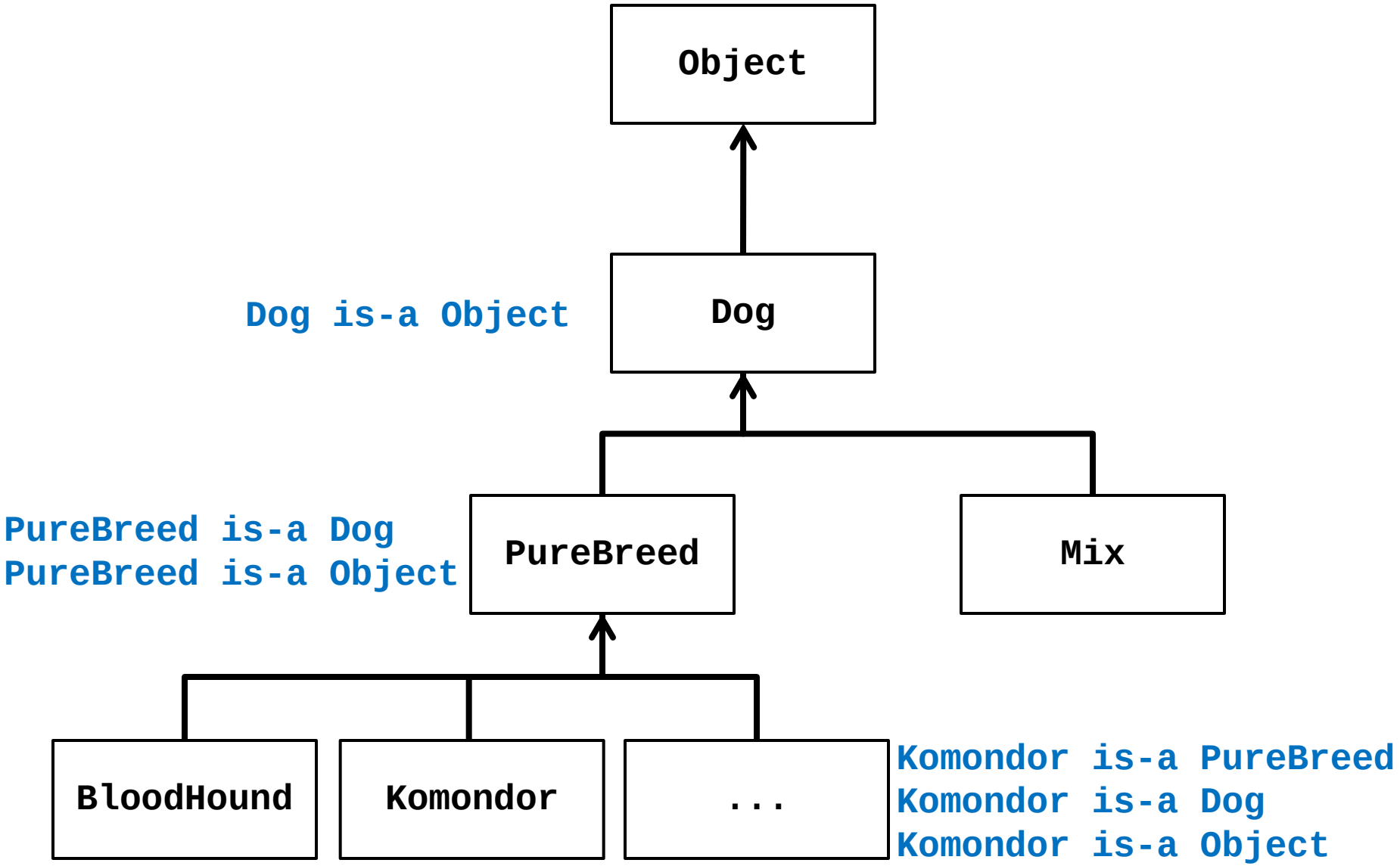
Inheritance

Notes Chapter 6 and AJ Chapters 7 and 8

Inheritance

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

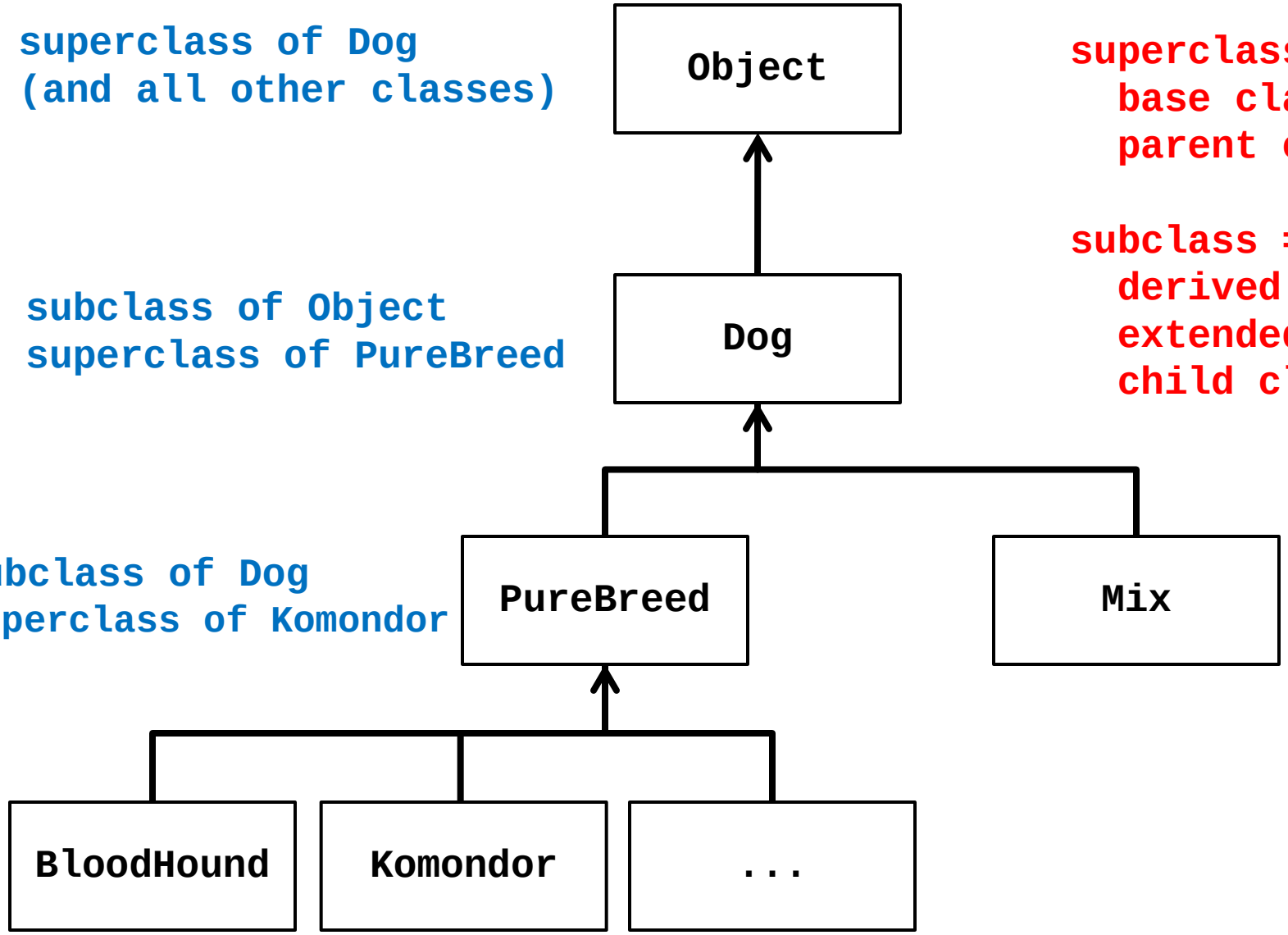
http://en.wikipedia.org/wiki/File:Komondor_delvin.jpg


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

superclass of Dog
(and all other classes)

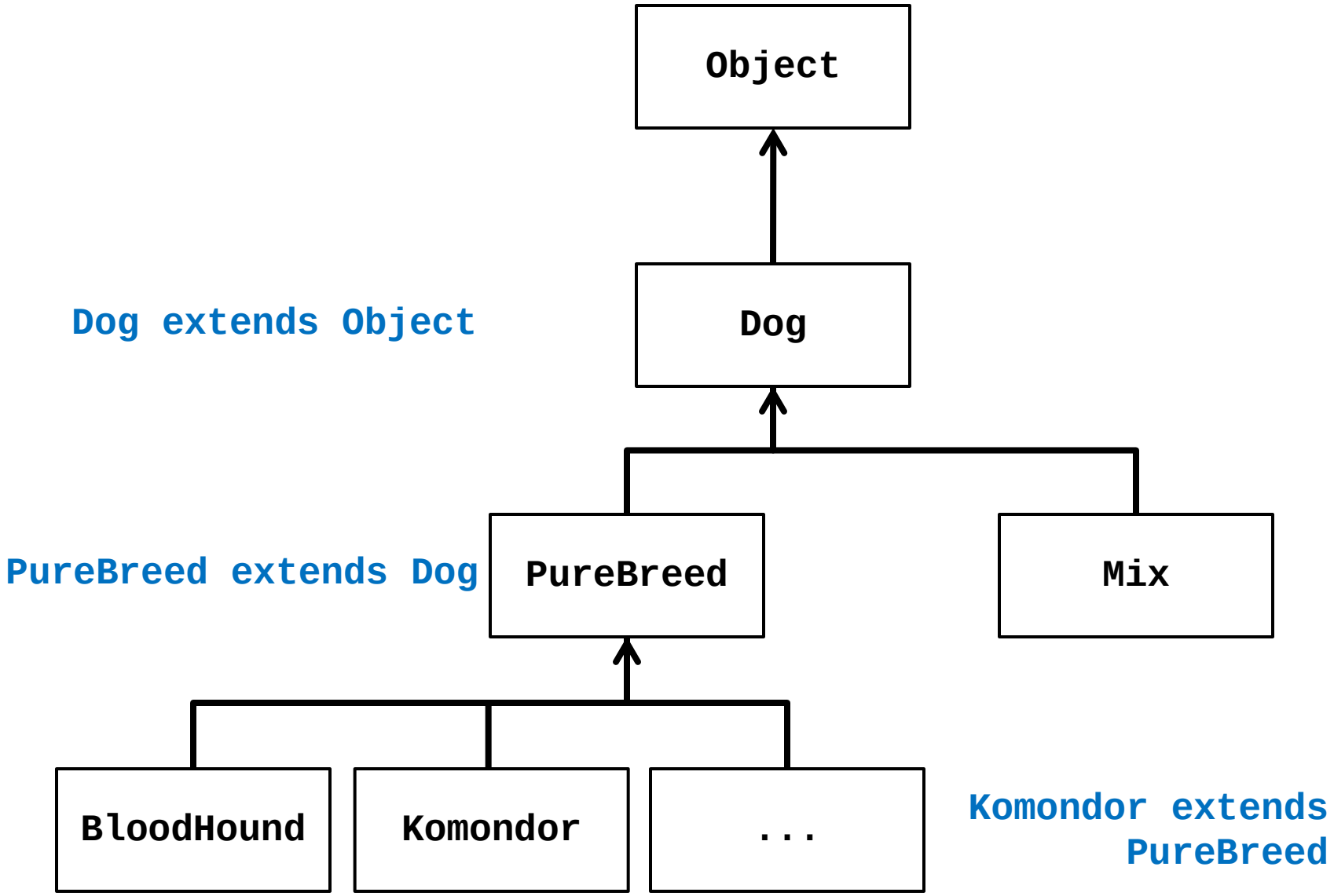
subclass of Object
superclass of PureBreed

subclass of Dog
superclass of Komondor



superclass ==
base class
parent class

subclass ==
derived class
extended class
child class


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 attributes and methods
  - the new class can re-define (override) its superclass methods

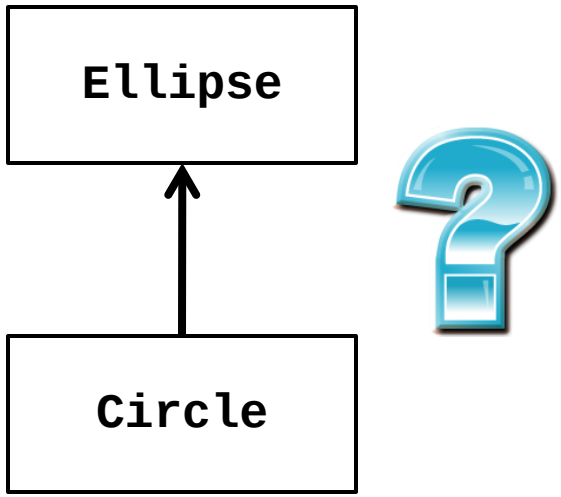
Is-A

- 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

- 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 attributes:
  - appearance, size, energy, grooming requirements, amount of exercise needed, protectiveness, compatibility with children, etc.
  - we will assume two attributes 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 attributes
- 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 (version 1)

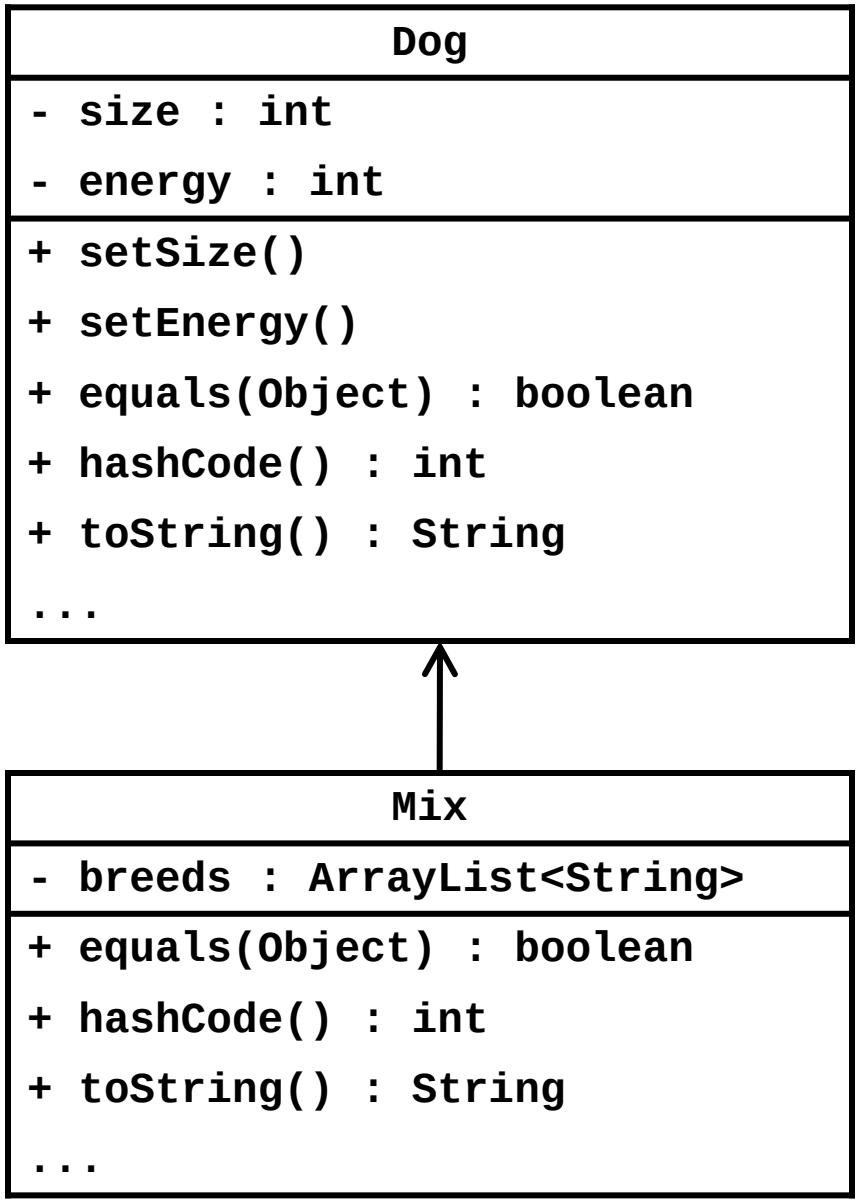
```
public final class Mix extends Dog
{ // no declaration of size or energy; inherited from Dog
  private ArrayList<String> breeds;

  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);
}
```


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
...

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

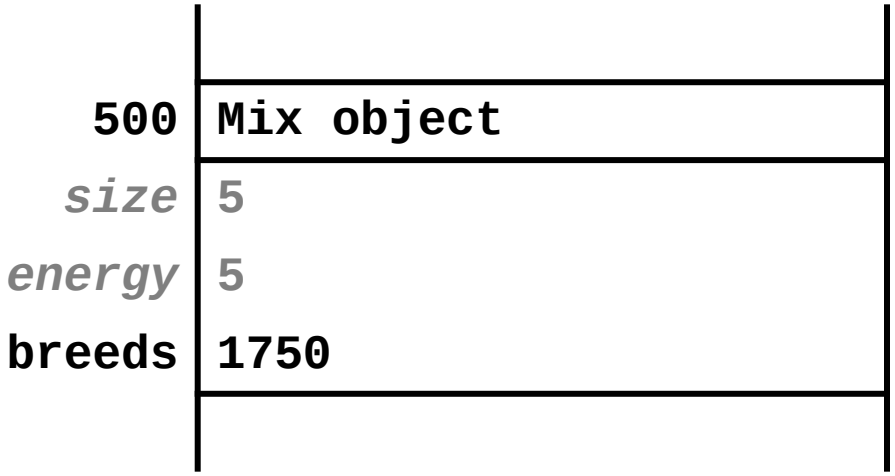
```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = 1;
  result = prime * result + this.getEnergy();
  result = prime * result + this.getSize();
  return result;
}
```

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



Mix UML Diagram