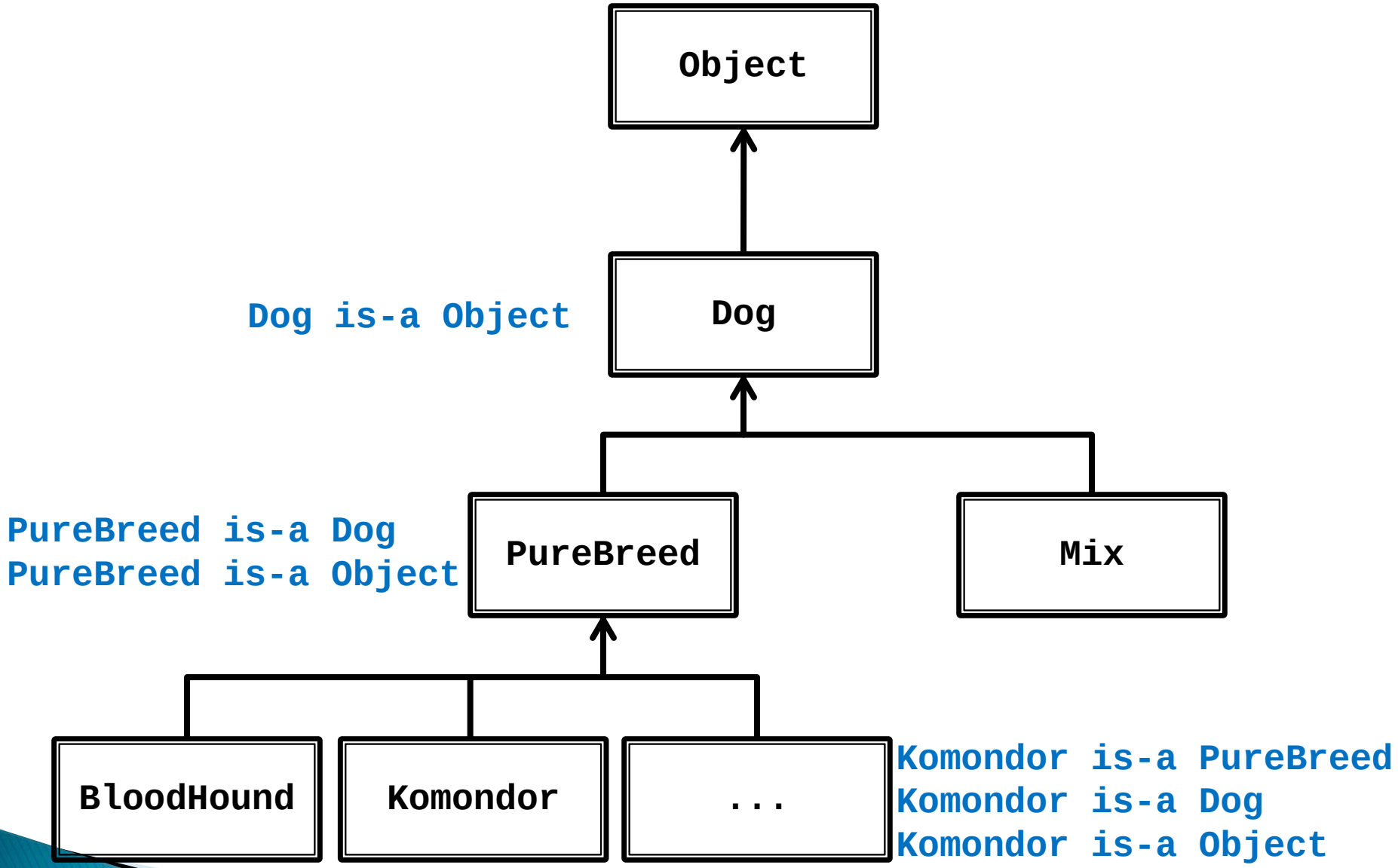
Inheritance

Based on slides by Prof. Burton Ma

Inheritance

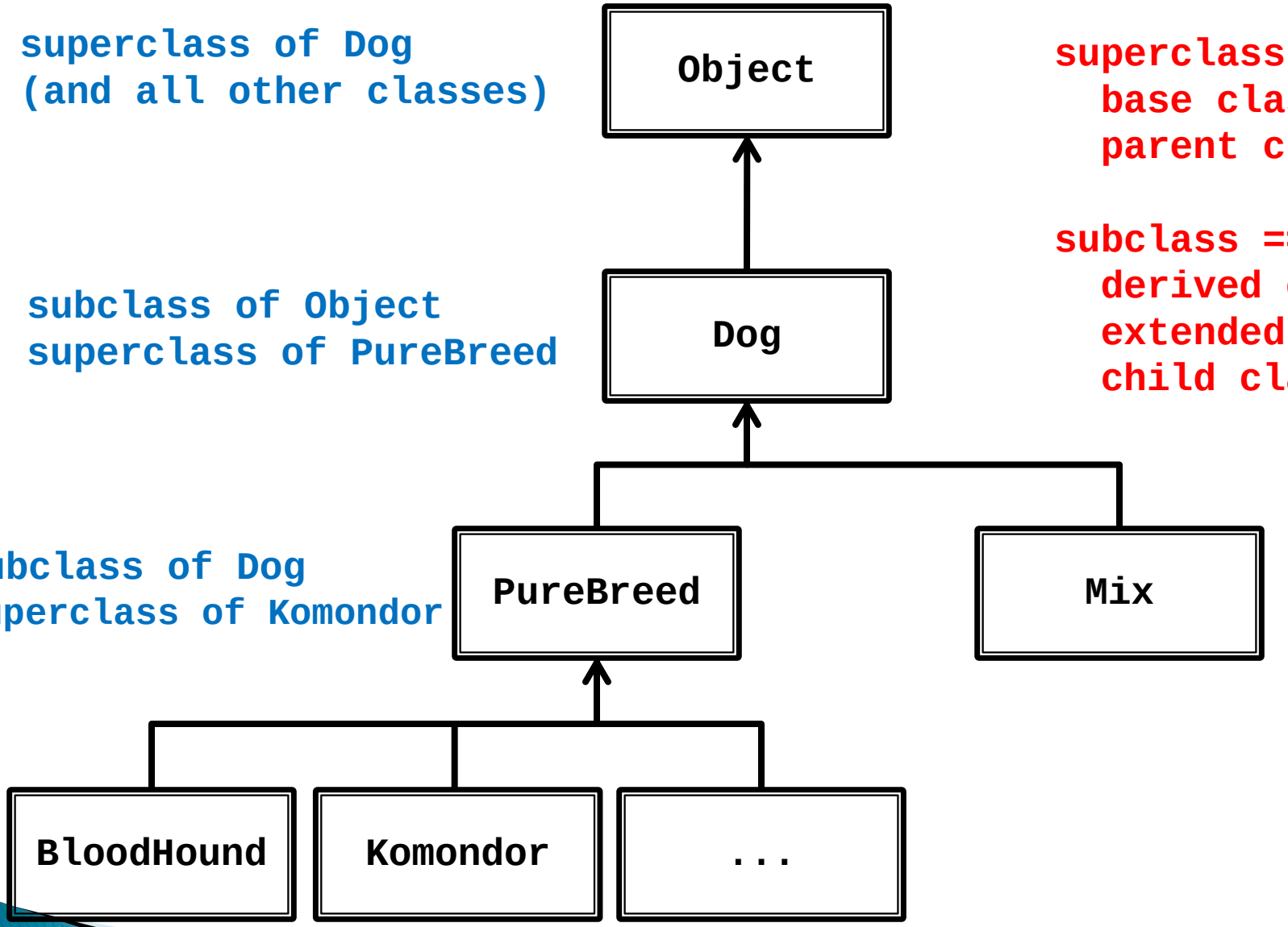
- You know a lot about an object by knowing its class
  - For example what is a Komondor?


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)

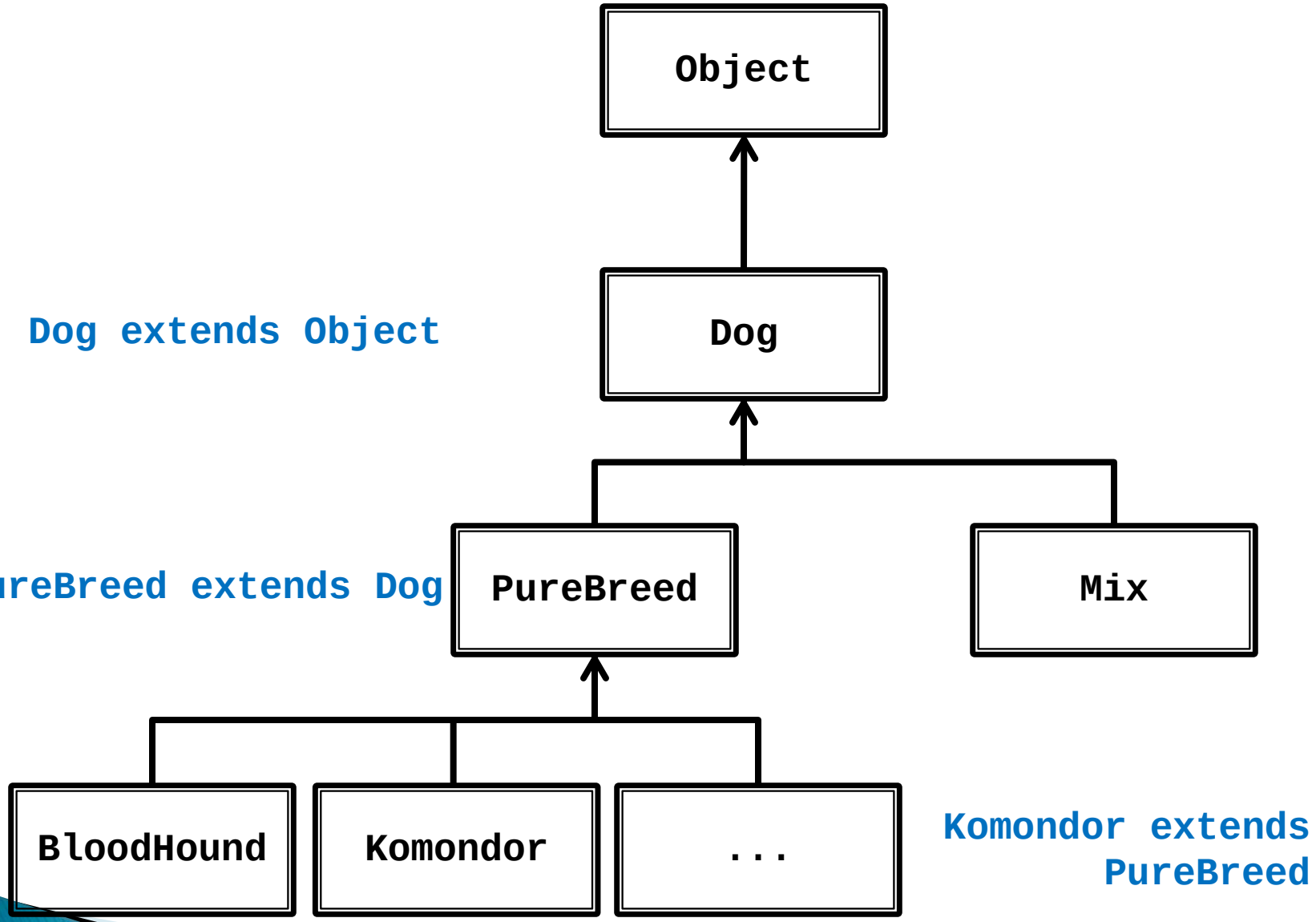
subclass of Object
superclass of PureBreed

subclass of Dog
superclass of Komondor



superclass ==
base class
parent class

subclass ==
derived class
extended class
child class


Object
Dog
Dog extends Object
PureBreed
PureBreed extends Dog
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

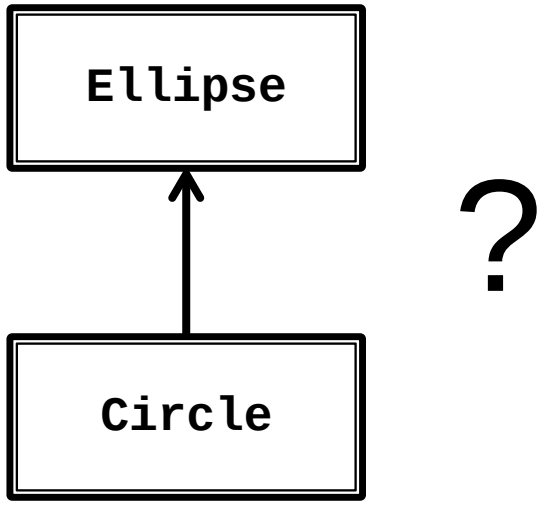
Is-A

- 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>();
  }
```

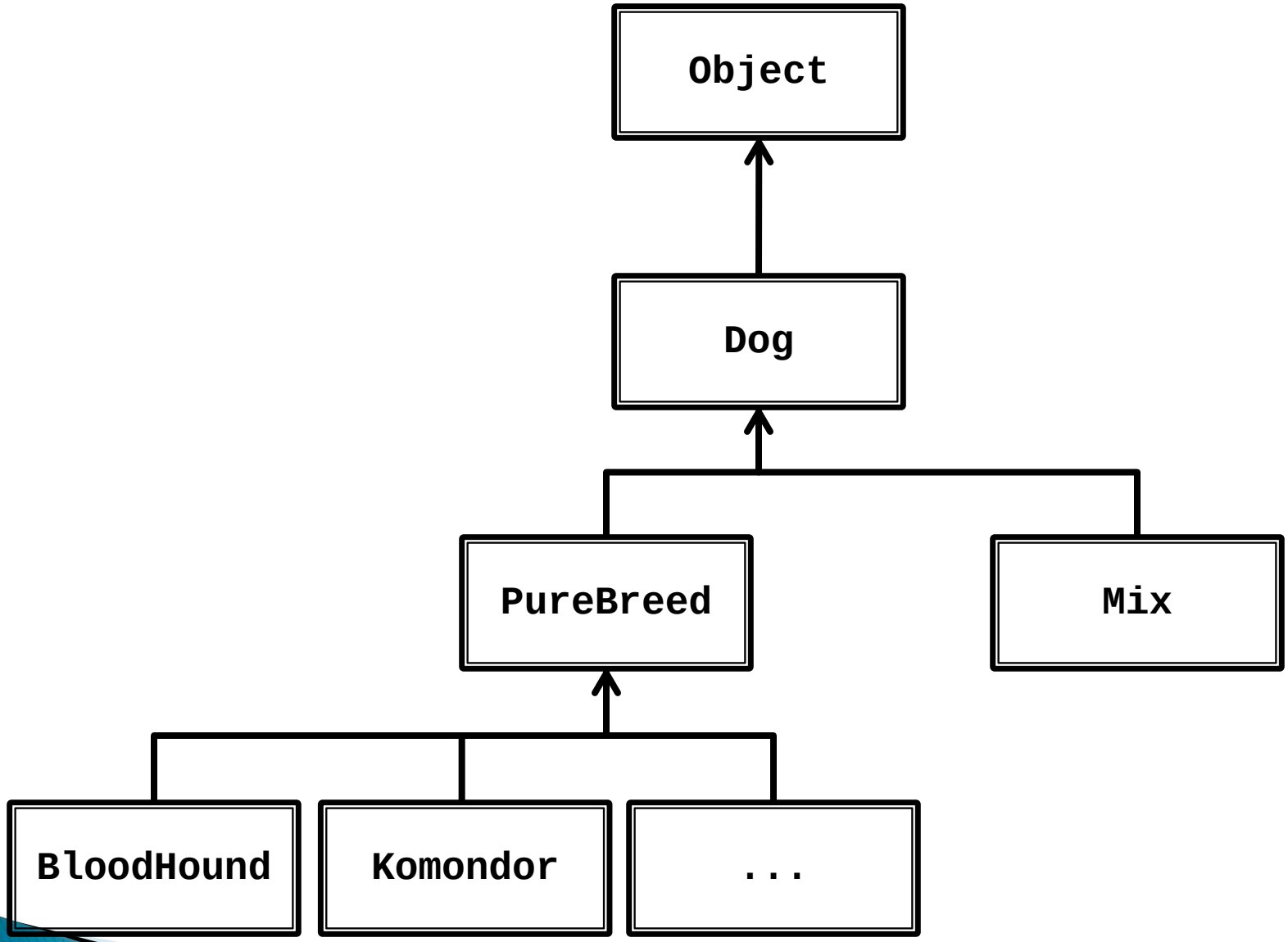
```
 public Mix(int size, int energy, ArrayList<String> breeds)
 { // call to a Dog constructor
   super(size, energy);
   this.breeds = new ArrayList<String>(breeds);
 }
}
```

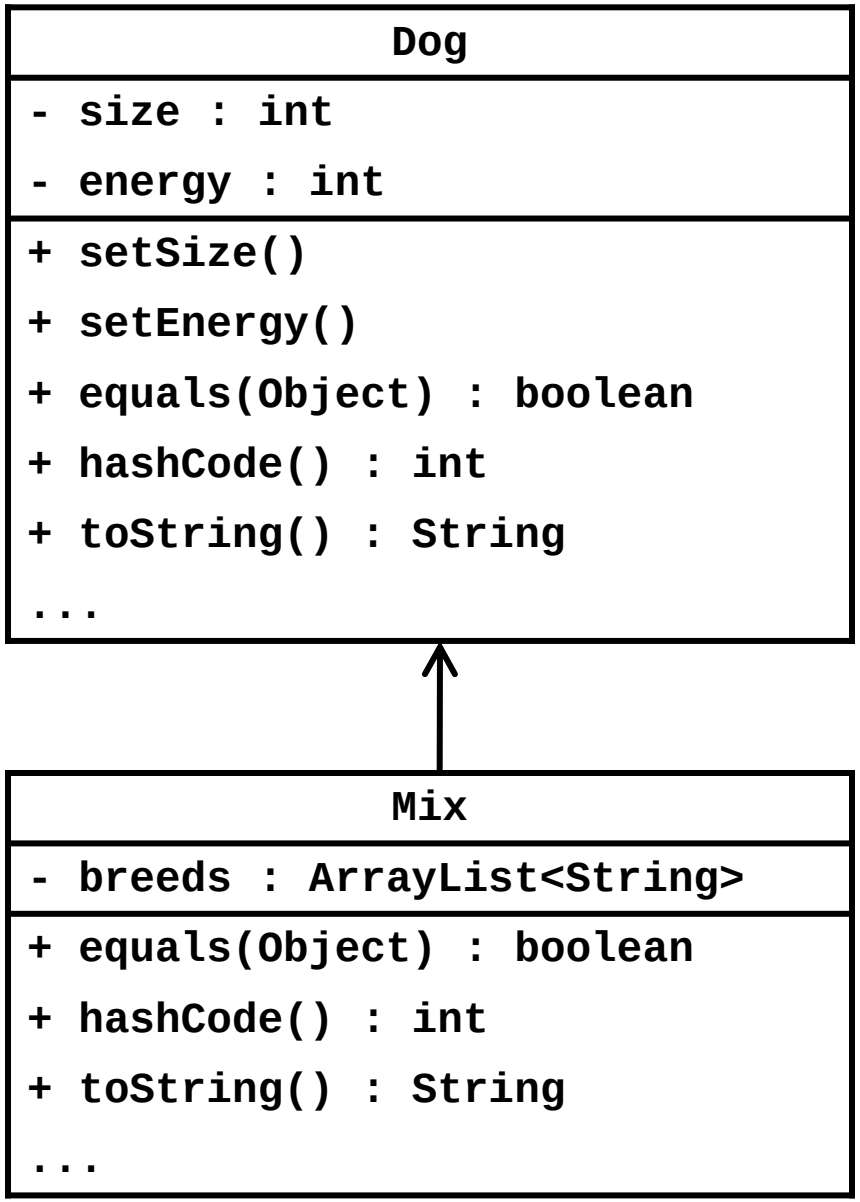
Mix (version 2)

```
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);
  }
}
```

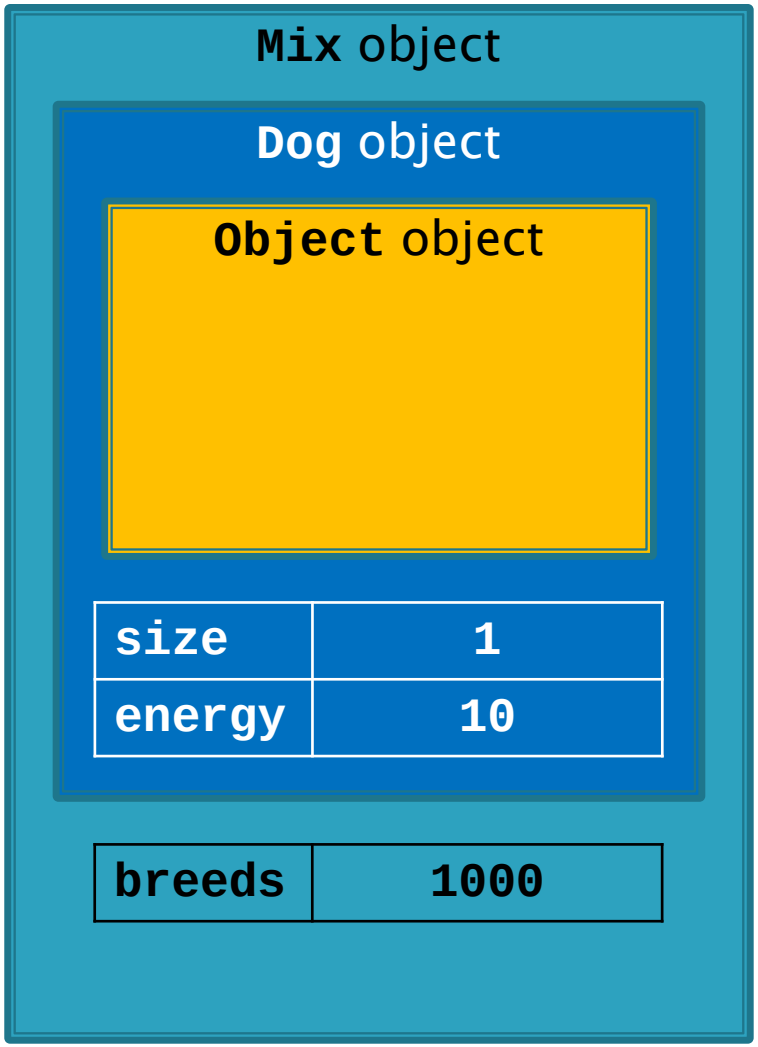
- Why is the constructor call to the superclass needed?
  - Because **`Mix`** is-a **`Dog`** and the **`Dog`** part of **`Mix`** needs to be constructed


Object
Dog
PureBreed
Mix
BloodHound
Komondor
...


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

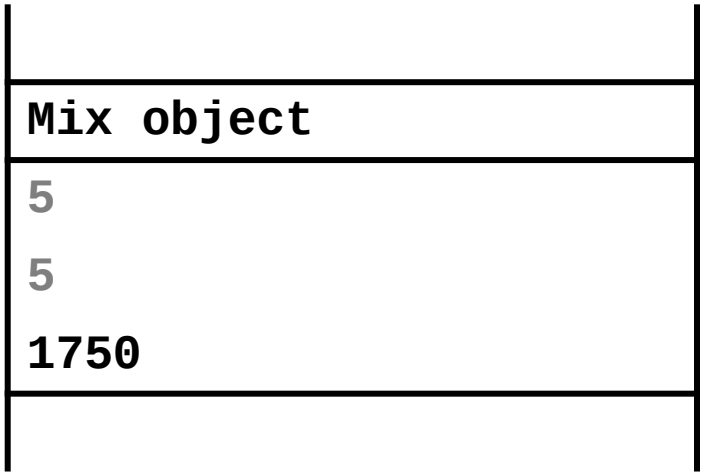
```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = 1;
  result = prime * result + this.getEnergy();
  result = prime * result + this.getSize();
  return result;
}
```

Mix hashCode

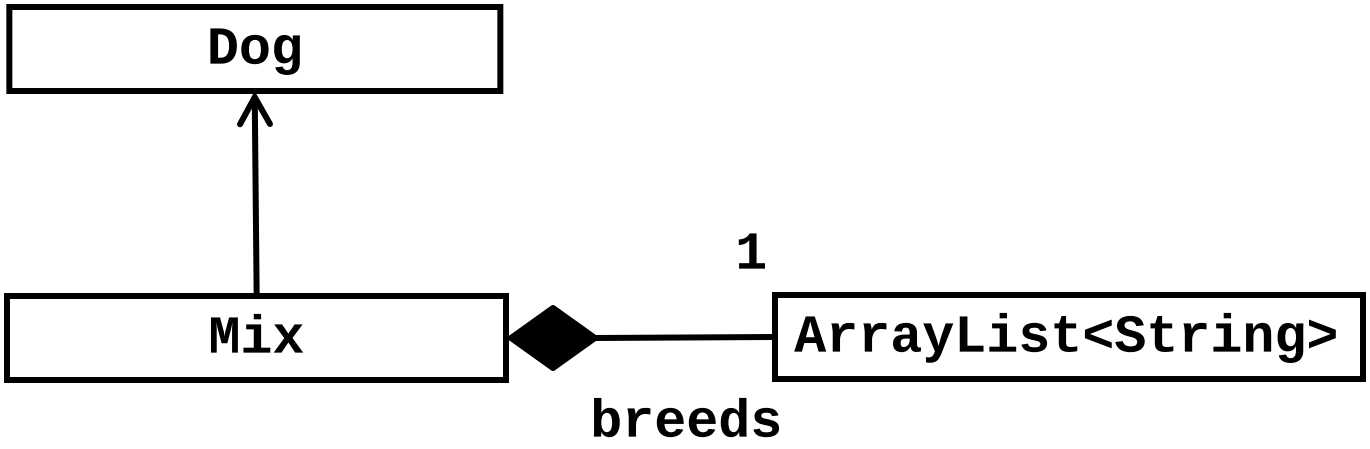
```
// similar to code generated by Eclipse
@Override public int hashCode()
{
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + this.breeds.hashCode();
  return result;
}
```

Mix Memory Diagram

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



Mix UML Diagram



Preconditions and Inheritance

- Precondition
  - What the method assumes to be true about the arguments passed to it
- Inheritance (is-a)
  - A subclass is supposed to be able to do everything its superclasses can do
- How do they interact?

Strength of a Precondition

- To strengthen a precondition means to make the precondition more restrictive

```
// Dog setEnergy
// 1. no precondition
// 2. 1 <= energy
// 3. 1 <= energy <= 10
public void setEnergy(int energy)
{ ... }
```

weakest precondition

strongest precondition

Preconditions on Overridden Methods

- A subclass can change a precondition on a method *but it must not strengthen the precondition*
  - A subclass that strengthens a precondition is saying that it cannot do everything its superclass can do

```
// Dog setEnergy
// assume non-final
// @pre. none

public
void setEnergy(int nrg)
{ // ... }
```

```
// Mix setEnergy
// bad : strengthen precond.
// @pre. 1 <= nrg <= 10

public
void setEnergy(int nrg)
{
  if (nrg < 1 || nrg > 10)
  { // throws exception }
  // ...
}
```

- Client code written for **Dog**s now fails when given a **Mix**

```
// client code that sets a Dog's energy to zero
public void walk(Dog d)
{
  d.setEnergy(0);
}
```

- Remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Postconditions and Inheritance

- Postcondition
  - What the method promises to be true when it returns
    - The method might promise something about its return value
      - “Returns size where size is between 1 and 10 inclusive"
    - The method might promise something about the state of the object used to call the method
      - “Sets the size of the dog to the specified size"
    - The method might promise something about one of its parameters
- How do postconditions and inheritance interact?

Strength of a Postcondition

- To strengthen a postcondition means to make the postcondition more restrictive

```
// Dog getSize
// 1. no postcondition
// 2. 1 <= this.size
// 3. 1 <= this.size <= 10
public int getSize()
{ ... }
```

weakest postcondition

strongest postcondition

Postconditions on Overridden Methods

- A subclass can change a postcondition on a method *but it must not weaken the postcondition*
  - A subclass that weakens a postcondition is saying that it cannot do everything its superclass can do

```
// Dog getSize
//
// @post. 1 <= size <= 10

public
int getSize()
{ // ... }
```

```
// Dogzilla getSize
// bad : weaken postcond.
// @post. 1 <= size

public
int getSize()
{ // ... }
```

Dogzilla: a made-up breed of dog that has no upper limit on its size

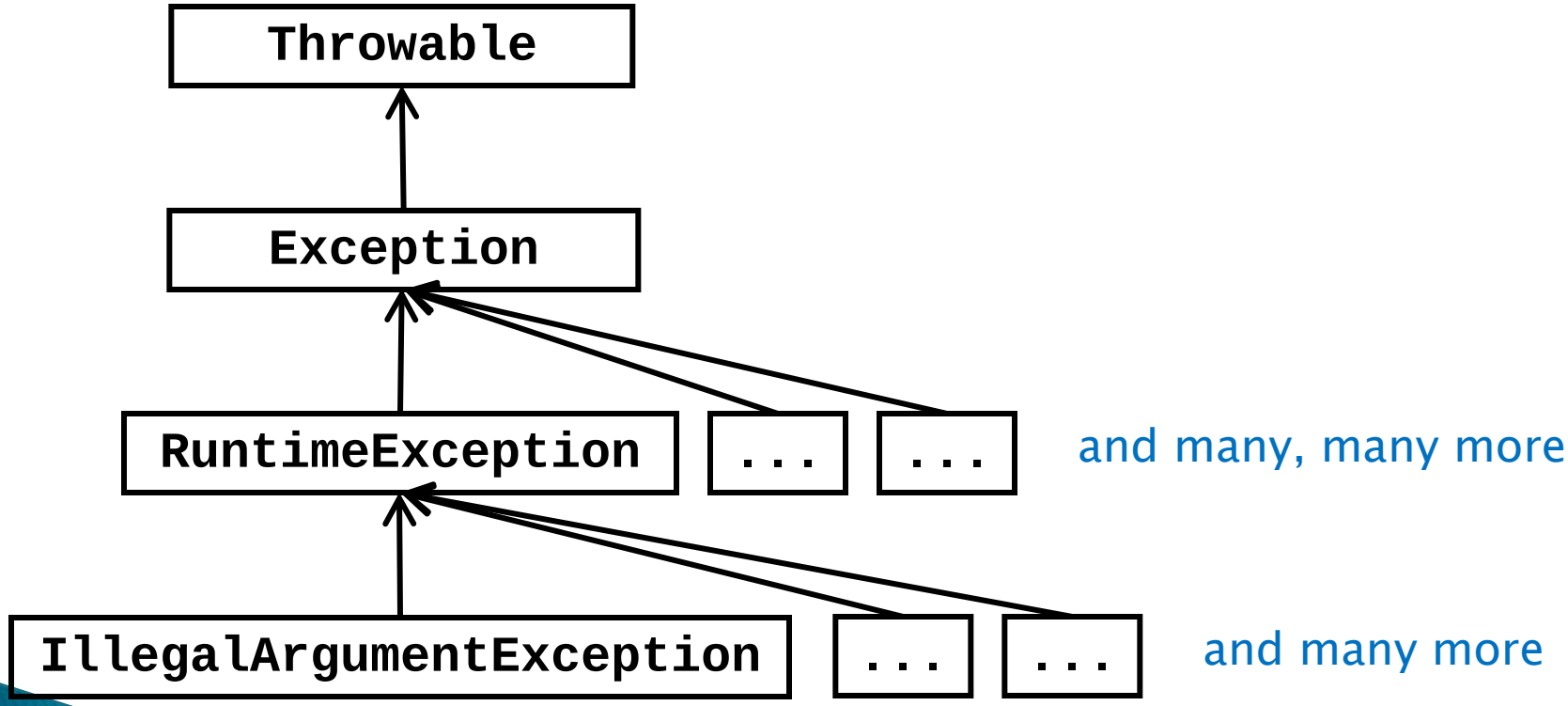
- Client code written for **Dog**s can now fail when given a **Dogzilla**

```
// client code that assumes Dog size <= 10
public String sizeToString(Dog d)
{
  int sz = d.getSize();
  String result = "";
  if (sz < 4)         result = "small";
  else if (sz < 7)    result = "medium";
  else if (sz <= 10) result = "large";
  return result;
}
```

- Remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

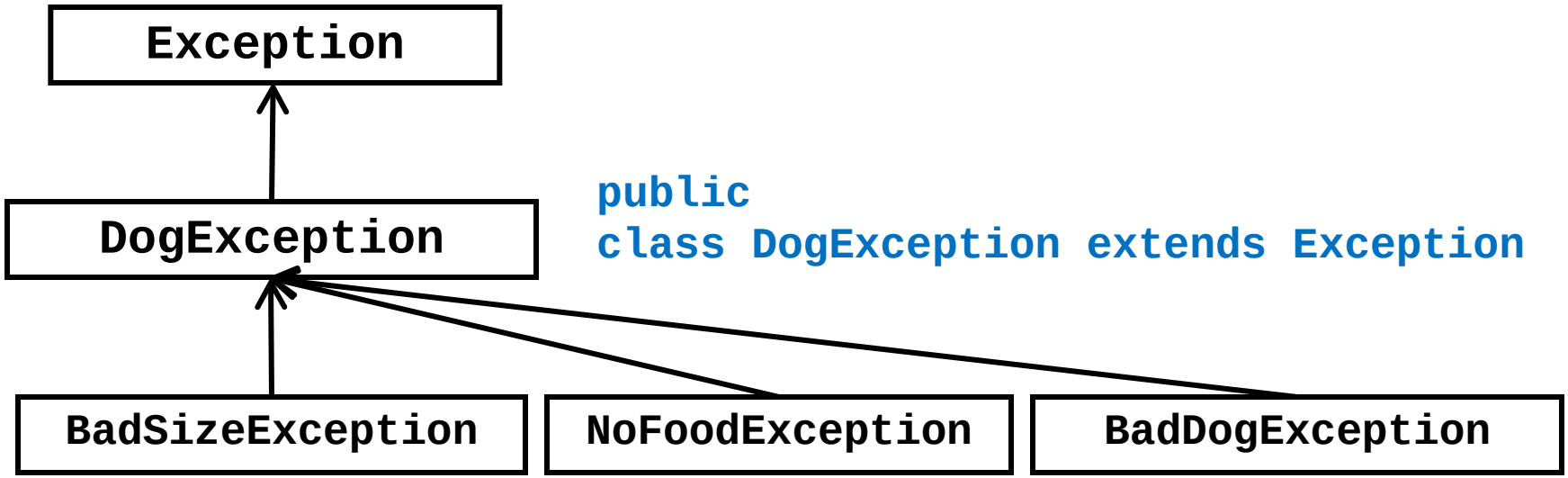
Exceptions

- All exceptions are objects that are subclasses of **`java.lang.Throwable`**



User Defined Exceptions

- You can define your own exception hierarchy
  - Often, you will subclass Exception




```
public
class DogException extends Exception
```

Exceptions and Inheritance

- A method that claims to throw an exception of type **X** is allowed to throw any exception type that is a subclass of **X**
  - This makes sense because exceptions are objects and subclass objects are substitutable for ancestor classes

```
// in Dog
public void someDogMethod() throws DogException
{
  // can throw a DogException, BadSizeException,
  //              NoFoodException, or BadDogException
}
```

- A method that overrides a superclass method that claims to throw an exception of type **X** must also throw an exception of type **X** or a subclass of **X**
  - Remember: a subclass promises to do everything its superclass does; if the superclass method claims to throw an exception then the subclass must also

```
// in Mix
@Override
public void someDogMethod() throws DogException
{
  // ...
}
```

Which are Legal?

- In Mix

```
@Override
public void someDogMethod() throws BadDogException
```

```
@Override
public void someDogMethod() throws Exception
```

```
@Override
public void someDogMethod()
```

```
@Override
public void someDogMethod()
      throws DogException, IllegalArgumentException
```

Inheritance Recap

- Inheritance allows you to create subclasses that are substitutable for their ancestors
  - Inheritance interacts with preconditions, postconditions, and exception throwing
- Subclasses
  - Inherit all non-private features
  - Can add new features
  - Can change the behaviour of non-final methods by *overriding* the parent method
  - Contain an instance of the superclass
    - Subclasses must construct the instance via a superclass constructor

Polymorphism

- Inheritance allows you to define a base class that has attributes and methods
  - Classes derived from the base class can use the public and protected base class attributes and methods
- Polymorphism allows the implementer to change the behaviour of the derived class methods

```
// client code
public void print(Dog d)
{
  System.out.println( d.toString() );
}

// later on...
Dog          fido = new Dog();
CockerSpaniel lady = new CockerSpaniel();
Mix          mutt = new Mix();
this.print(fido);          Dog toString
this.print(lady);          CockerSpaniel toString
this.print(mutt);          Mix toString
```

- Notice that **`fido`**, **`lady`**, and **`mutt`** were declared as **`Dog`**, **`CockerSpaniel`**, and **`Mutt`**
- What if we change the declared type of **`fido`**, **`lady`**, and **`mutt`** ?

```
// client code
public void print(Dog d) {
  System.out.println( d.toString() );
}

// later on...
Dog          fido = new Dog();
Dog          lady = new CockerSpaniel();
Dog          mutt = new Mix();
this.print(fido);        Dog toString
this.print(lady);        CockerSpaniel toString
this.print(mutt);        Mix toString
```

- What if we change the **`print`** method parameter type to **`Object`** ?

```
// client code
public void print(Object obj) {
  System.out.println( obj.toString() );
}

// later on...
Dog          fido = new Dog();
Dog          lady = new CockerSpaniel();
Dog          mutt = new Mix();
this.print(fido);          Dog toString
this.print(lady);          CockerSpaniel toString
this.print(mutt);          Mix toString
this.print(new Date());    Date toString
```

Late Binding

- Polymorphism requires *late binding* of the method name to the method definition
  - Late binding means that the method definition is determined at run-time

```
obj.toString()
```

run-time type of the instance **obj**

non-static method

Declared vs Run-time type

```
Dog lady = new CockerSpaniel();
```

declared type

run-time or actual type

- The declared type of an instance determines what methods can be used

```
Dog lady = new CockerSpaniel();
```

  - The name **`lady`** can only be used to call methods in **`Dog`**
  - **`lady.someCockerSpanielMethod()`** won't compile

- The actual type of the instance determines what definition is used when the method is called

```
Dog lady = new CockerSpaniel();
```

  - **`lady.toString()`** uses the **`CockerSpaniel`** definition of **`toString`**

Abstract Classes

- Sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - E.g. you might want to know what a **Dog**'s bark sounds like but the sound of the bark depends on the breed of the dog
    - You want to add the method bark to **Dog** but only the subclasses of **Dog** can implement **bark**
  - E.g. you might want to know the breed of a **Dog** but only the subclasses have information about the breed
    - You want to add the method **getBreed** to **Dog** but only the subclasses of **Dog** can implement **getBreed**

Abstract Classes

- Sometimes you will find that you want the API for a base class to have a method that the base class cannot define
  - E.g. you might want to know the breed of a **`Dog`** but only the subclasses have information about the breed
    - You want to add the method **`getBreed`** to **`Dog`** but only the subclasses of **`Dog`** can implement **`getBreed`**

- If the base class has methods that only subclasses can define *and* the base class has attributes common to all subclasses then the base class should be abstract
  - If you have a base class that just has methods that it cannot implement then you probably want an interface
- Abstract :
  - (Dictionary definition) existing only in the mind

- In Java an abstract class is a class that you cannot make instances of

- An abstract class provides a partial definition of a class
  - The subclasses complete the definition
- An abstract class can define attributes and methods
  - Subclasses inherit these
- An abstract class can define constructors
  - Subclasses can call these
- An abstract class can declare abstract methods
  - Subclasses must define these (unless the subclass is also abstract)

Abstract Methods

- An abstract base class can declare, but not define, zero or more abstract methods

```
public abstract class Dog
{
  // attributes, ctors, regular methods

  public abstract String getBreed();
}
```

- The base class is saying "all **Dog**s can provide a **String** describing the breed, but only the subclasses know enough to implement the method"

Abstract Methods

- The non-abstract subclasses must provide definitions for all abstract methods
  - Consider **`getBreed`** in **`Mix`**

```
public class Mix extends Dog
{ // stuff from before...
  @Override public String getBreed()
    {
      if(this.breeds.isEmpty()) {
        return "mix of unknown breeds";
    }
    StringBuffer b = new StringBuffer();
    b.append("mix of");
    for(String breed : this.breeds)
    {
      b.append(" " + breed);
    }
  return b.toString();
}
```

PureBreed

- A purebreed dog is a dog with a single breed
  - One **`String`** attribute to store the breed
- Note that the breed is determined by the subclasses
  - The class **`PureBreed`** cannot give the **`breed`** attribute a value
  - But it can implement the method **`getBreed`**
- The class **`PureBreed`** defines an attribute common to all subclasses and it needs the subclass to inform it of the actual breed
  - **`PureBreed`** is also an abstract class

```
public abstract class PureBreed extends Dog
{
  private String breed;

  public PureBreed(String breed)
  {
    super();
    this.breed = breed;
  }

  public PureBreed(String breed, int size, int energy)
  {
    super(size, energy);
    this.breed = breed;
  }
```

```
  @Override public String getBreed()
  {
    return this.breed;
  }

}
```

Subclasses of PureBreed

- The subclasses of **`PureBreed`** are responsible for setting the breed
  - Consider **`Komondor`**

Komondor

```
public class Komondor extends PureBreed
{
  private final String BREED = "komondor";

  public Komondor()
  {
    super(BREED);
  }

  public Komondor(int size, int energy)
  {
    super(BREED, size, energy);
  }

  // other Komondor methods...
}
```

Static Attributes and Inheritance

- Static attributes behave the same as non-static attributes in inheritance
  - Public and protected static attributes are inherited by subclasses, and subclasses can access them directly by name
  - Private static attributes are not inherited and cannot be accessed directly by name
    - But they can be accessed/modified using public and protected methods

Static Attributes and Inheritance

- The important thing to remember about static attributes and inheritance
  - There is only one copy of the static attribute shared among the declaring class and all subclasses
- Consider trying to count the number of **Dog** objects created by using a static counter

```
// the wrong way to count the number of Dogs created
public abstract class Dog
{
  // other attributes...
  static protected int numCreated = 0;

  Dog()
  {
    // ...
    Dog.numCreated++;
  }

  public static int getNumberCreated()
  {
    return Dog.numCreated;
  }

  // other contructors, methods...
}
```

protected, not private, so that subclasses can modify it directly

```
// the wrong way to count the number of Dogs created
public class Mix extends Dog
{
  // attributes...

  Mix()
  {
    super();
    Mix.numCreated++;
  }

  // other contructors, methods...
}
```

```
// too many dogs!

public class TooManyDogs
{
  public static void main(String[] args)
  {
    Mix mutt = new Mix();
    System.out.println( Mix.getNumberCreated() );
  }
}
```

prints 2

What Went Wrong?

- There is only one copy of the static attribute shared among the declaring class and all subclasses
  - **`Dog`** declared the static attribute
  - **`Dog`** increments the counter everytime its constructor is called
  - **`Mix`** inherits and shares the single copy of the attribute
  - **`Mix`** constructor correctly calls the superclass constructor
    - Which causes **`numCreated`** to be incremented by **`Dog`**
  - **`Mix`** constructor then incorrectly increments the counter

Counting Dogs and Mixes

- Suppose you want to count the number of **`Dog`** instances and the number of **`Mix`** instances
  - **`Mix`** must also declare a static attribute to hold the count
    - Somewhat confusingly, **`Mix`** can give the counter the same name as the counter declared by **`Dog`**

```
public class Mix extends Dog
{
  // other attributes...
  private static int numCreated = 0;  // bad style

  public Mix()
  {
    super();      // will increment Dog.numCreated
    // other Mix stuff...
    numCreated++; // will increment Mix.numCreated
  }

  // ...
```

Hiding Attributes

- Note that the **`Mix`** attribute **`numCreated`** has the same name as an attribute declared in a superclass
  - Whenever **`numCreated`** is used in **`Mix`**, it is the **`Mix`** version of the attribute that is used
- If a subclass declares an attribute with the same name as a superclass attribute, we say that the subclass attribute hides the superclass attribute
  - Considered bad style because it can make code hard to read and understand
    - Should change **`numCreated`** to **`numMixCreated`** in **`Mix`**

Static Methods and Inheritance

- There is a big difference between calling a static method and calling a non-static method when dealing with inheritance
- *There is no dynamic dispatch on static methods*

```
public abstract class Dog
{
  private static int numCreated = 0;
  public static int getNumCreated()
  {
    return Dog.numCreated;
  }
}
public class Mix
{
  private static int numMixCreated = 0;
  public static int getNumCreated()
  {
    return Mix.numMixCreated;
  }
}
public class Komondor
{
  private static int numKomondorCreated = 0;
  public static int getNumCreated()
  {
    return Komondor.numKomondorCreated;
  }
}
```

notice no @Override

notice no @Override

```
public class WrongCount
{
  public static void main(String[] args)
  {
    Dog mutt = new Mix();
    Dog shaggy = new Komondor();
    System.out.println( mutt.getNumCreated() );
    System.out.println( shaggy.getNumCreated() );
    System.out.println( Mix.getNumCreated() );
    System.out.println( Komondor.getNumCreated() );
  }
}
```

prints 2

2

1

1

What's Going On?

- *There is no dynamic dispatch on static methods*

- Because the declared type of **`mutt`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

- Because the declared type of **`shaggy`** is **`Dog`**, it is the **`Dog`** version of **`getNumCreated`** that is called

Hiding Methods

- Notice that **`Mix.getNumCreated`** and **`Komondor.getNumCreated`** work as expected
- If a subclass declares a static method with the same name as a superclass static method, we say that the subclass static method hides the superclass static method
  - *You cannot override a static method, you can only hide it*
  - Hiding static methods is considered bad form because it makes code hard to read and understand

- The client code in **`WrongCount`** illustrates two cases of bad style, one by the client and one by the implementer of the **`Dog`** hierarchy
  1. The client should not have used an instance to call a static method
  2. The implementer should not have hidden the static method in **`Dog`**

Interfaces

- Recall that you typically use an abstract class when you have a superclass that has attributes and methods that are common to all subclasses
  - The abstract class provides a partial implementation that the subclasses must complete
  - Subclasses can only inherit from a single superclass
- If you want classes to support a common API then you probably want to define an interface

Interfaces

- In Java an *interface* is a reference type (similar to a class)
- An interface says what methods an object must have and what the methods are supposed to do
  - I.e., an interface is an API

Interfaces

- An interface can contain *only*
  - Constants
  - Method signatures
  - Nested types (ignore for now)

- There are no method bodies
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces

Interfaces Already Seen

access—either public or package-private (blank)

interface name

```
public interface Comparable<T>
{
  int compareTo(T o);
}
```

Interfaces Already Seen

```
public interface Iterable<T>
{
  Iterator<T> iterator();
}
```

access—either public or package-private (blank)

interface name

parent interfaces

```
public interface Collection<E> extends Iterable<E>
{
  boolean add(E e);
  void    clear();
  boolean contains(Object o);
  // many more method signatures...
}
```

Interfaces Already Seen

```
public interface List<E> extends Collection<E>
{
  boolean add(E e);
  void    add(int index, E element);
  boolean addAll(Collection<? extends E> c);
  // many more method signatures...
}
```

Creating an Interface

- Decide on a name
- Decide what methods you need in the interface

- This is harder than it sounds because...
  - Once an interface is released and widely implemented, it is almost impossible to change
    - If you change the interface, all classes implementing the interface must also change

Function Interface

- In mathematics, a real-valued scalar function of one real scalar variable maps a real value to another real value

$$y = f(x)$$

Creating an Interface

- Decide on a name
  - **`DoubleToDoubleFunction`**
- Decide what methods you need in the interface
  - **`double   at(double x)`**
  - **`double[] at(double[] x)`**

Creating an Interface

```
public interface DoubleToDoubleFunction
{
  double   at(double x);
  double[] at(double[] x);
}
```

Classes that Implement an Interface

- A class that implements an interface says so by using the **`implements`** keyword
  - Consider the function $f(x) = x^2$

```
public class Square implements DoubleToDoubleFunction
{
  public double at(double x)
  {
    return x * x;
  }

  public double[] at(double[] x)
  {
    double[] result = new double[x.length];
    for (int i = 0; i < x.length; i++)
    {
      result[i] = x[i] * x[i];
    }
    return result;
  }
}
```

Implementing Multiple Interfaces

- Unlike inheritance where a subclass can extend only one superclass, a class can implement as many interfaces as it needs to

```
public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>,
             RandomAccess,
             Cloneable,
             Serializable
```

superclass

interfaces