Aggregation and Composition

[notes Chapter 4]

Privacy Leaks

- a privacy leak occurs when a class exposes a reference to a non-public field (that is not a primitive or immutable)
  - given a class **X** that is a composition of a **Y**

```
public class X {
  private Y y;
  // …
}
```

these are all examples of privacy leaks

```
public X(Y y) {
  this.y = y;
}
```

```
public X(X other) {
  this.y = other.y;
}
```

```
public Y getY() {
  return this.y;
}
```

```
public void setY(Y y) {
  this.y = y;
}
```

Consequences of Privacy Leaks

- a privacy leak allows some other object to control the state of the object that leaked the field
  - the object state can become inconsistent
    - example: if a `CreditCard` exposes a reference to its expiry `Date` then a client could set the expiry date to before the issue date

Consequences of Privacy Leaks

- a privacy leak allows some other object to control the state of the object that leaked the field
  - it becomes impossible to guarantee class invariants
    - example: if a **`Period`** exposes a reference to one of its **`Date`** objects then the end of the period could be set to before the start of the period

Consequences of Privacy Leaks

- a privacy leak allows some other object to control the state of the object that leaked the field
  - composition becomes broken because the object no longer owns its attribute
    - when an object "dies" its parts may not die with it

Recipe for Immutability

- the recipe for immutability in Java is described by Joshua Bloch in the book *Effective Java**

1. Do not provide any methods that can alter the state of the object
2. Prevent the class from being extended
3. Make all fields **`final`**
4. Make all fields **`private`**
5. Prevent clients from obtaining a reference to any mutable fields

revisit when we talk about inheritance

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);
```

100	`client` invocation
dates	200
d1	500
d2	600
d3	700
	...
200	`ArrayList` object
	500
	600
	700

Test Your Knowledge

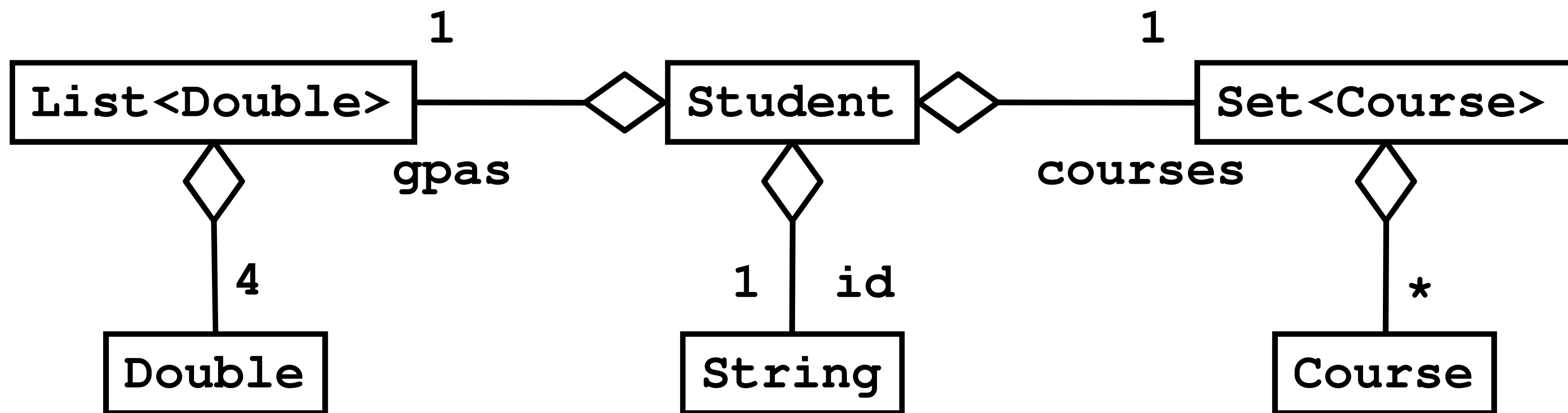
1. What does the following print?

```
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`**?

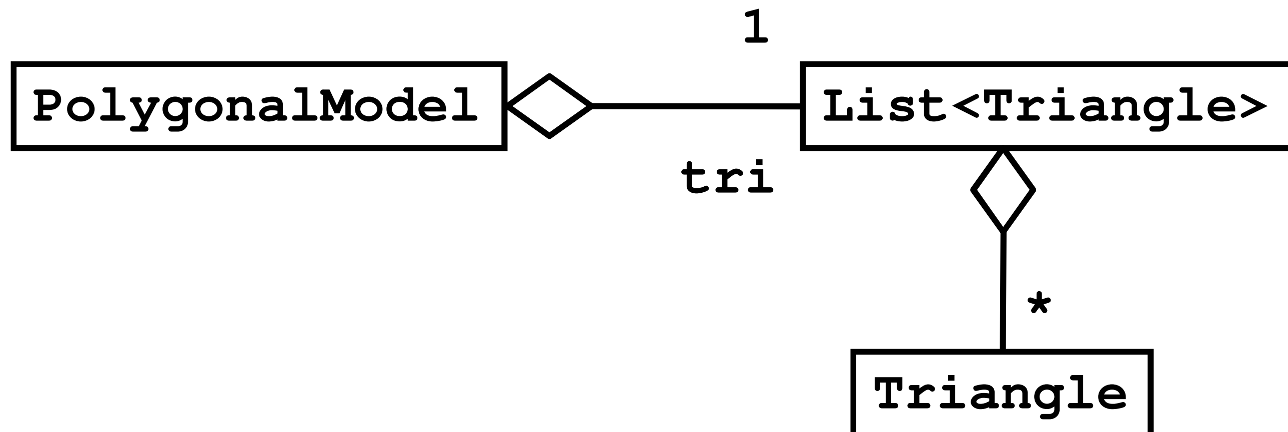
Student Class (from notes)

- 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 **`Triangle`**s
  - 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

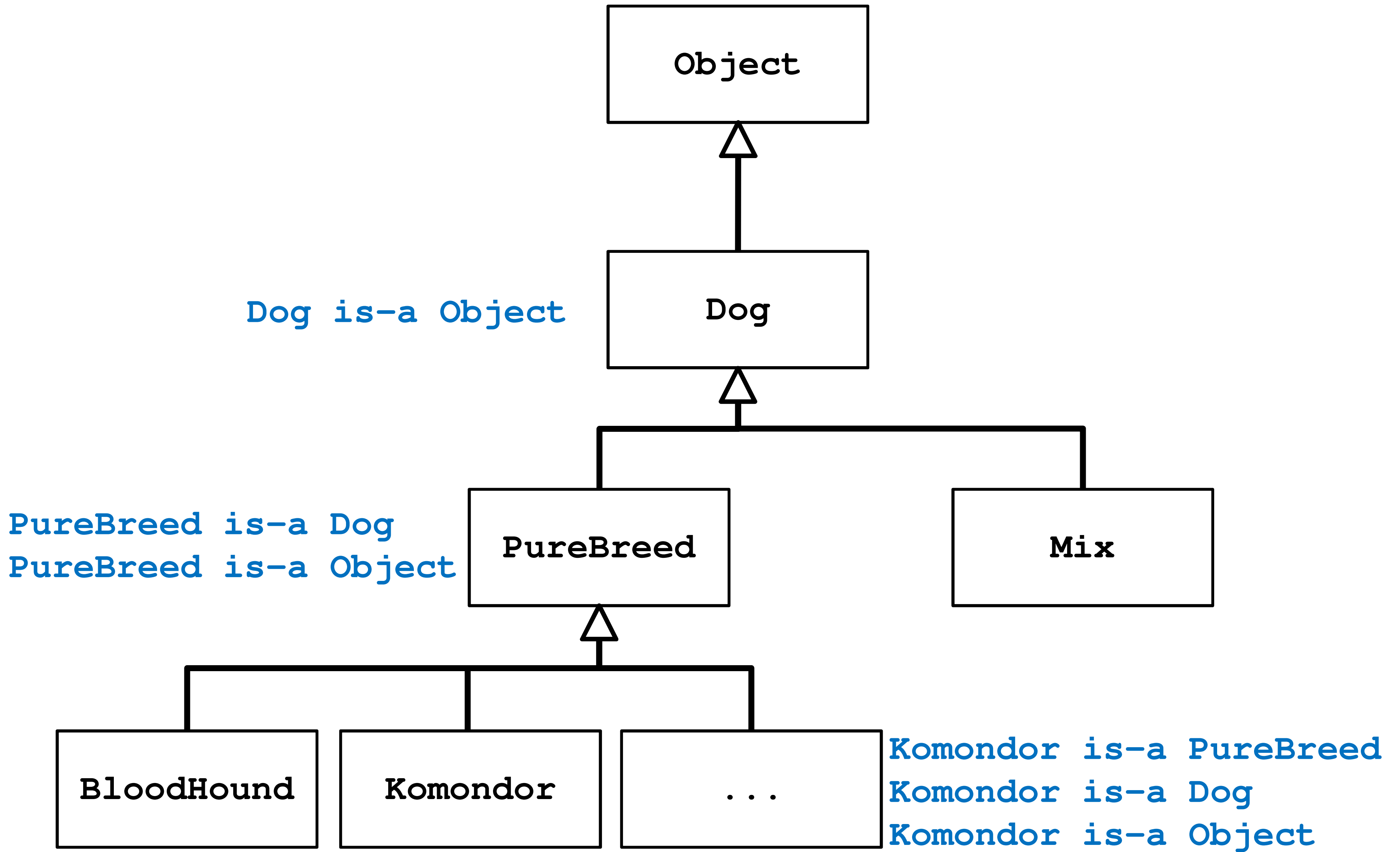
Inheritance

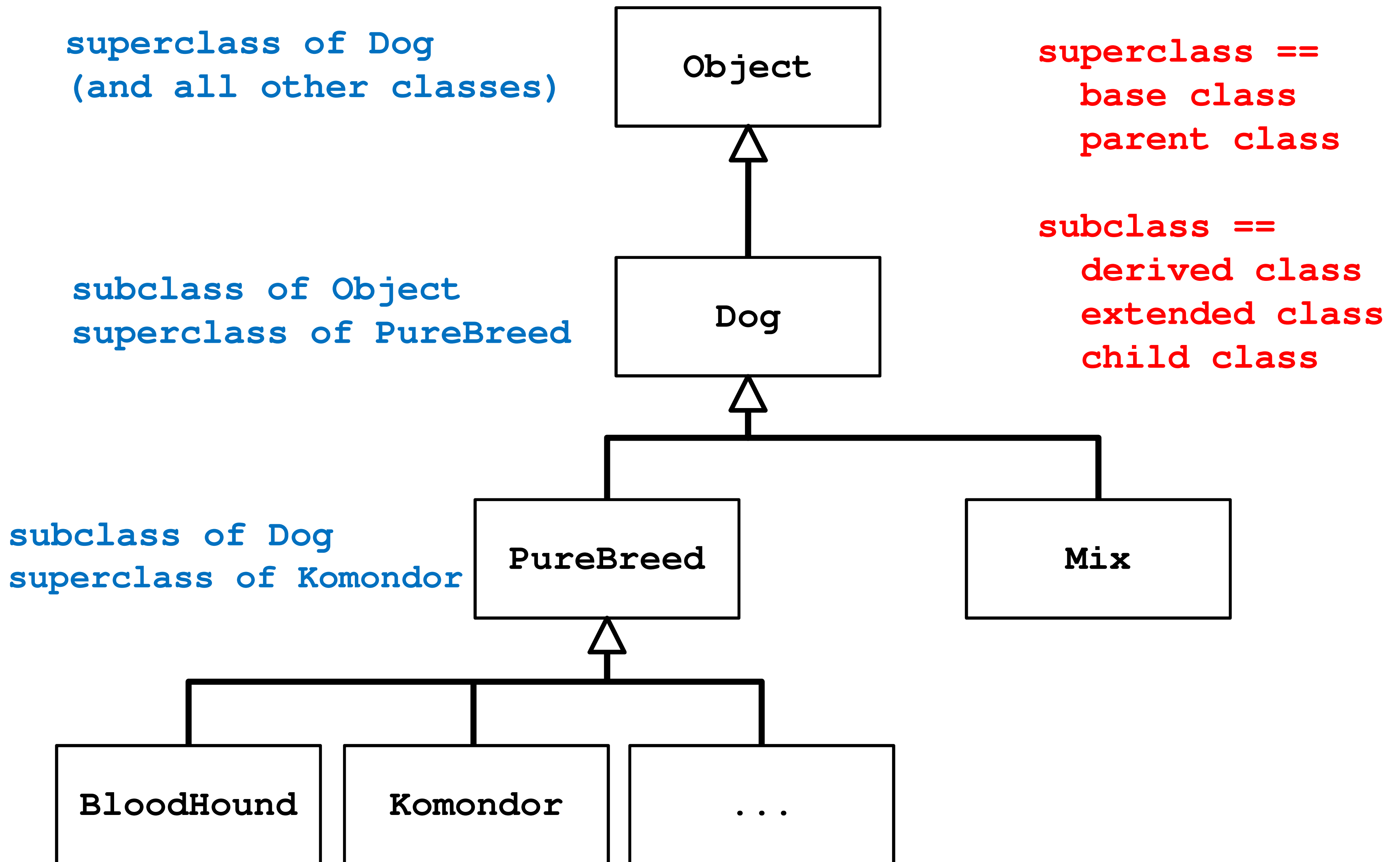
Notes Chapter 6

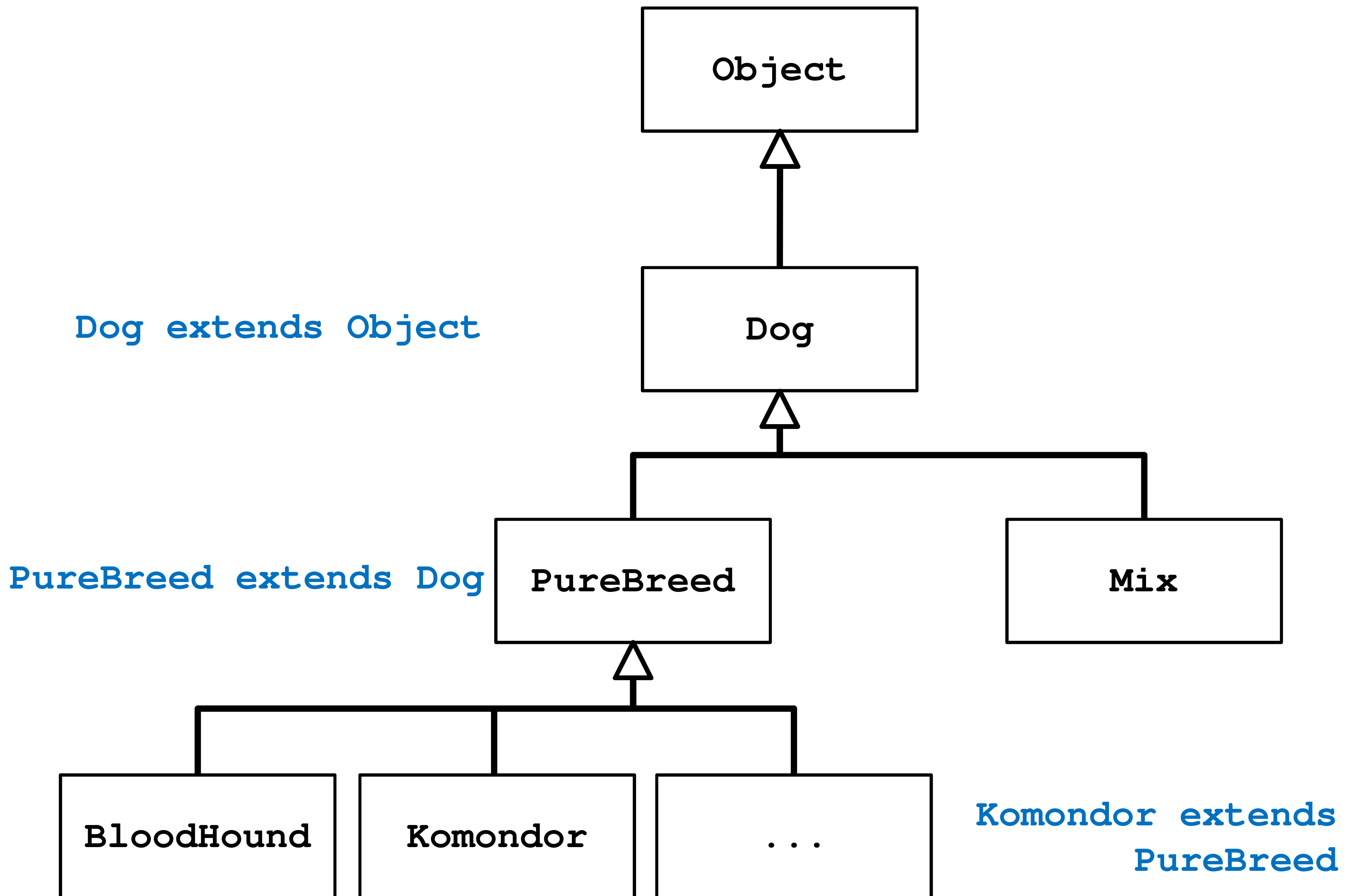
Inheritance

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

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


Object
Dog
Dog is-a Object
PureBreed
Mix
PureBreed is-a Dog
PureBreed is-a Object
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
Dog extends Object
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

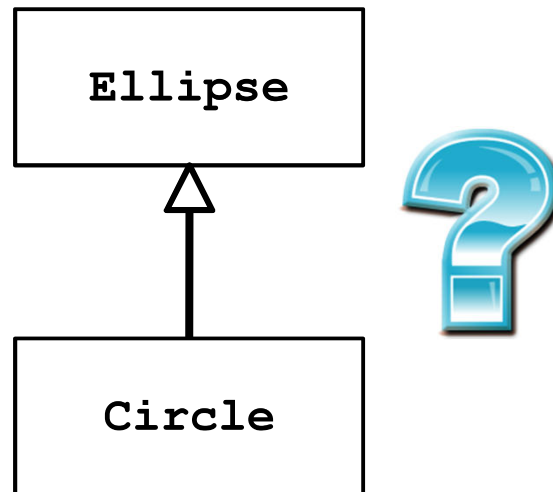
Is-A

- 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

```java
// 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…

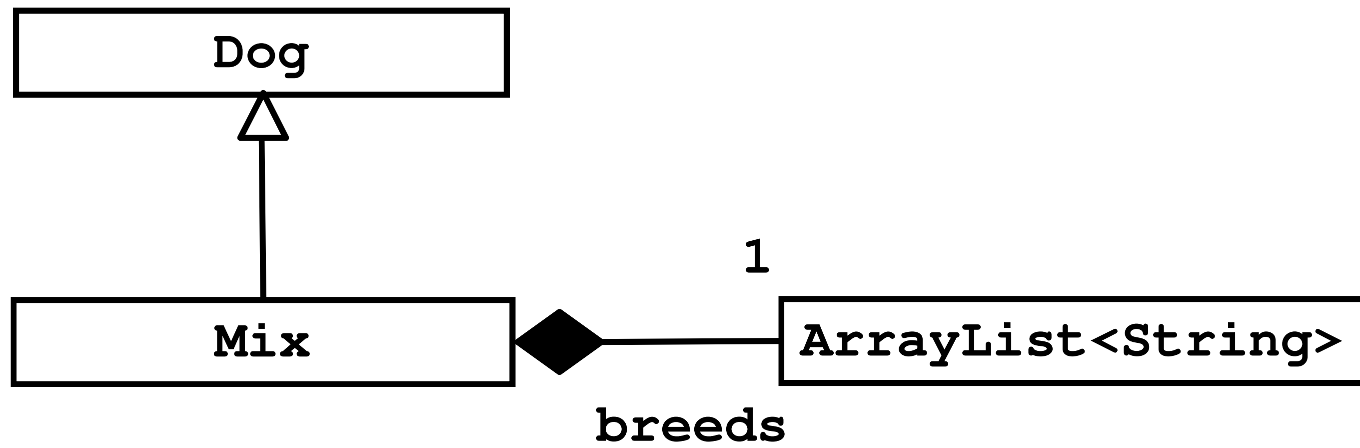
What is a Subclass?

- 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>();
  }
```

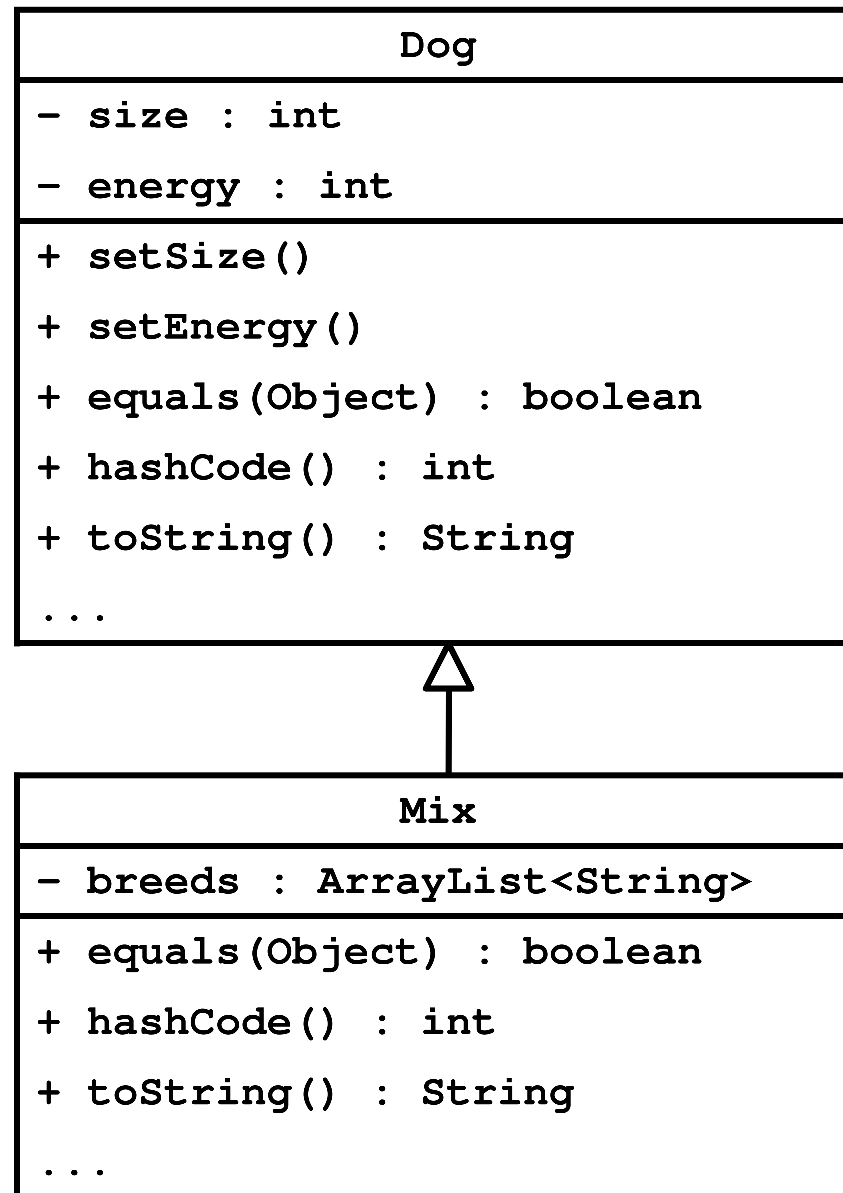
```
public Mix(int size, int energy,
           ArrayList<String> breeds)
{ // call to a Dog constructor
  super(size, energy);
  this.breeds = new ArrayList<String>(breeds);
}
```

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

```java
@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

500	**Mix object**
size	5
energy	5
breeds	**1750**

Inheritance (Part 2)

Preconditions and Inheritance

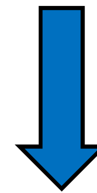
- precondition
  - what the method assumes to be true about the arguments passed to it
- inheritance (is-a)
  - a subclass is supposed to be able to do everything its superclasses can do
- how do they interact?

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()
{ ... }
```

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

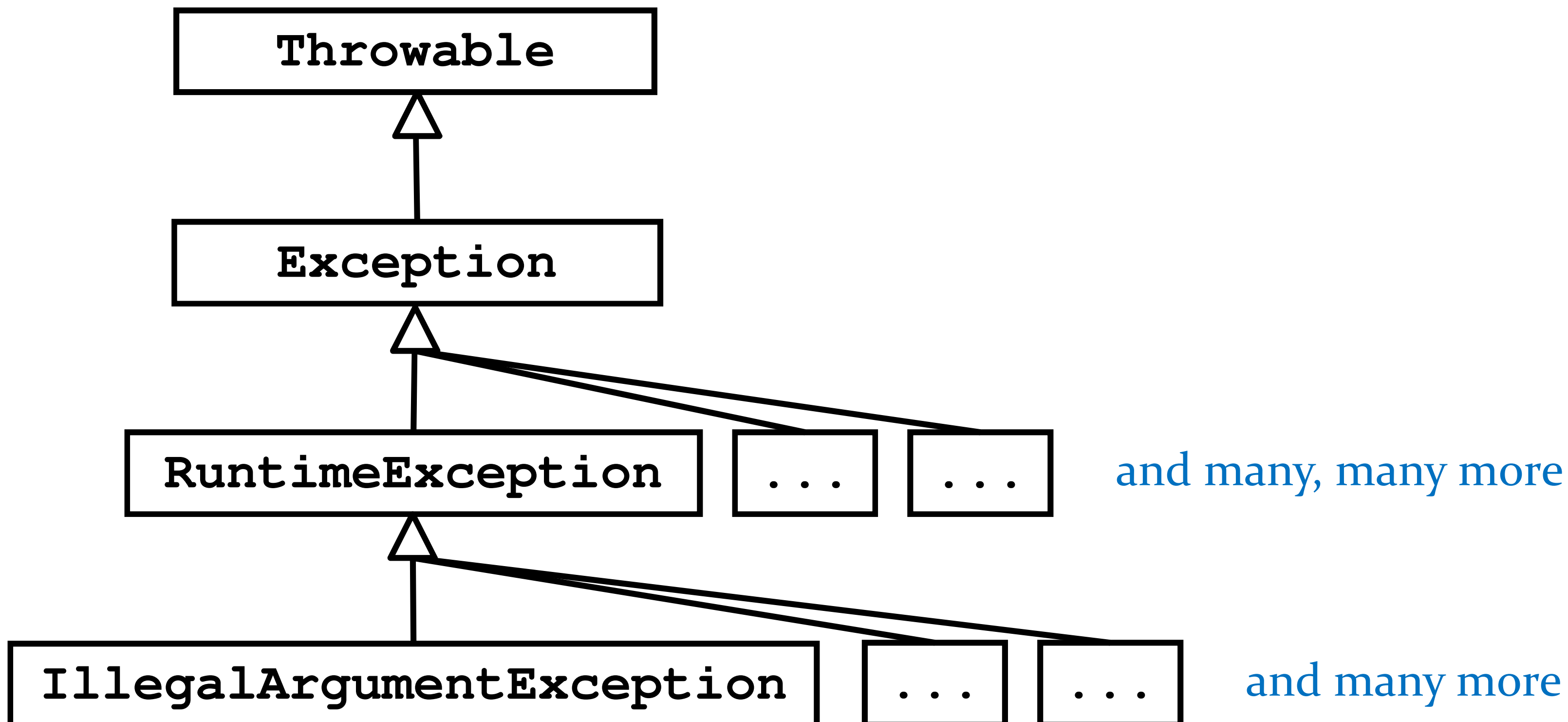
- client code written for **`Dog`**s can now fail when given a **`Dogzilla`**

```
// 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

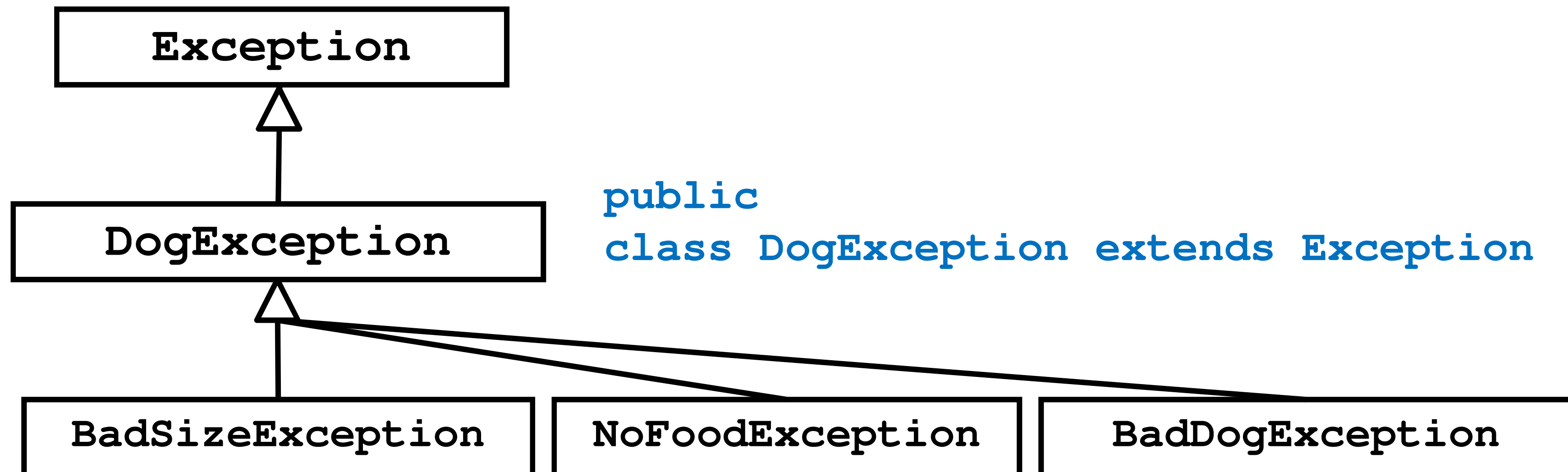
Exceptions

- 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



Exceptions and Inheritance

- a method that claims to throw a *checked* exception of type **X** is allowed to throw any checked 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 a checked exception of type **`X`** can also claim to throw a checked exception of type **`X`** or a subclass of **`X`**
  - remember: a subclass is substitutable for the parent type

```
// 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
```

Review

1. Inheritance models the ______ relationship between classes.
2. Dog is a ______ of Object.
3. Dog is a ______ of Mix.
4. Can a Dog instance do everything a Mix instance can?
5. Can a Mix instance do everything a Dog instance can?
6. Is a Dog instance substitutable for a Mix instance?
7. Is a Mix instance substitutable for a Dog instance?

8. Can a subclass use the private fields of its superclass?
9. Can a subclass use the private methods of its superclass?
10. Suppose you have a class X that you do not want anyone to extend. How do you enforce this?
11. Suppose you have an immutable class X. Someone extends X to make it mutable. Is this legal?
12. What do you need to do to enforce immutability?

13. Suppose you have a class Y that extends X.
    a. Does each Y instance have a X instance inside of it?
    b. How do you construct the X subobject inside of the Y instance?
    c. What syntax is used to call the superclass constructor?
    d. What is constructed first–the X subobject or the Y object?
    e. Suppose Y introduces a brand new method that needs to call a public method in X named xMethod. How does the new Y method call xMethod?
    f. Suppose Y overrides a public method in X named xMethod. How does the overriding Y method call xMethod?

14. Suppose you have a class Y that extends X. X has a method with the following precondition:

```
@pre. value must be a multiple of 2
```

If Y overrides the method which of the following are acceptable preconditions for the overriding method:

a. `@pre. value must be a multiple of 2`
b. `@pre. value must be odd`
c. `@pre. value must be a multiple of 2 and must be less than 100`
d. `@pre. value must be a multiple of 10`
e. `@pre. none`

14. Suppose you have a class Y that extends X. X has a method with the following postcondition:

`@return - A String of length 10`

If Y overrides the method which of the following are acceptable postconditions for the overriding method:

a. **`@return - A String of length 9 or 10`**
b. **`@return - The String "weimaraner"`**
c. **`@return - An int`**
d. **`@return - The same String returned by toString`**
e. **`@return - A random String of length 10`**

15. Suppose Dog toString has the following Javadoc:

```
/*
 * Returns a string representation of a dog.
 * The string is the size of the dog followed by a
 * a space followed by the energy.
 * @return The string representation of the dog.
 */
```

Does this affect subclasses of Dog?

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

Puzzle 3

- Write the class **`Enigma`**, which extends **`Object`**, so that the following program prints false:

```
public class Conundrum
{
  public static void main(String[] args)
  {
    Enigma e = new Enigma();
    System.out.println( e.equals(e) );
  }
}
```

- You must not override **`Object.equals()`**

[*Java Puzzlers* by Joshua Block and Neal Gaffer]

Polymorphism

- inheritance allows you to define a base class that has fields and methods
  - classes derived from the base class can use the public and protected base class fields 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() );
}                            Dog toString
                             CockerSpaniel toString
                             Mix toString
// later on...
Dog           fido = new Dog();
CockerSpaniel lady = new CockerSpaniel();
Mix           mutt = new Mix();
this.print(fido);
this.print(lady);
this.print(mutt);
```

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

- 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);
this.print(lady);
this.print(mutt);
this.print(new Date());
```

```
Dog toString
CockerSpaniel toString
Mix toString
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

non-static method

```
obj.toString()
```

run-time type of the instance `obj`

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`

Inheritance (Part 3)

Abstract Classes

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`**

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`**

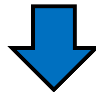
- if the base class has methods that only subclasses can define *and* the base class has fields common to all subclasses then the base class should be abstract
  - if you have a base class that just has methods that it cannot implement then you probably want an interface
- abstract :
    - (dictionary definition) existing only in the mind
- in Java an abstract class is a class that you cannot make instances of
  - e.g. http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html

- an abstract class provides a partial definition of a class
  - the subclasses complete the definition

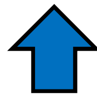
- an abstract class can define fields and methods
  - subclasses *inherit* these
- an abstract class can define constructors
  - subclasses *must call* these
- an abstract class can declare abstract methods
  - subclasses *must define* these (unless the subclass is also abstract)

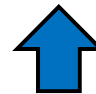
Abstract Methods

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



```
public abstract class Dog
{
  // fields, 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`** field to store the breed
- note that the breed is determined by the subclasses
  - the class **`PureBreed`** cannot give the **`breed`** field a value
  - but it can implement the method **`getBreed`**
- the class **`PureBreed`** defines an field 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...
}
```

Inheritance (Part 4)

Static Features; Interfaces

Static Fields and Inheritance

- static fields behave the same as non-static fields in inheritance
  - public and protected static fields are inherited by subclasses, and subclasses can access them directly by name
  - private static fields are not inherited and cannot be accessed directly by name
    - but they can be accessed/modified using public and protected methods

Static Fields and Inheritance

- the important thing to remember about static fields and inheritance
  - *there is only one copy of the static field shared among the declaring class and all subclasses*

- consider trying to count the number of `Dog` objects created by using a static counter

```java
// the wrong way to count the number of Dogs created
public abstract class Dog {
  // other fields...
  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

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

  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 field shared among the declaring class and all subclasses
  - `Dog` declared the static field
  - `Dog` increments the counter everytime its constructor is called
  - `Mix` inherits and shares the single copy of the field
  - `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 field 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 fields...
  private static int numCreated = 0;  // bad style

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

  // ...
```

Hiding Fields

- note that the **`Mix`** field **`numCreated`** has the same name as an field declared in a superclass
  - whenever **`numCreated`** is used in **`Mix`**, it is the **`Mix`** version of the field that is used

- if a subclass declares an field with the same name as a superclass field, we say that the subclass field hides the superclass field
  - 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*
  - therefore, you cannot override a static method

```
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

Interfaces

- recall that you typically use an abstract class when you have a superclass that has fields 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