Interfaces

Interfaces

- in its most common form, a Java interface is a declaration (but *not* an implementation) of an API
- in its most common form, an interface is made up of public abstract methods
  - an abstract method is a method that has an API but does not have an implementation
- consider an interface for mathematical functions of the form $y = f(x)$

```java
import java.util.List;

public interface Function {

    /**
     * Evaluate the function at x.
     *
     * @param x the value at which to evaluate the function
     * @return the value of the function evaluated at x
     */
    public double eval(double x);

    /**
     * Evaluate the function at each value of x in the given list.
     *
     * @param x a list of values at which to evaluate the function
     * @return the list of values of the function evaluated at the given
     * values of x
     */
    public List<Double> eval(List<Double> x);
}
```

semicolon, and no method body

semicolon, and no method body

Interfaces

- notice that the interface declares which methods exist and specifies the contract of the methods
  - but it does *not* specify *how* the methods are implemented
- the method implementations are defined by classes that implement the interface
- consider the functions:
  - $y = x^2$
  - $y = \frac{1}{x}$
  - $y = \frac{4}{\pi}\left(\sum_{n=1,3,5\ldots}^{n_{max}} \frac{\sin(n\pi x)}{n}\right)$

```java
public class Square implements Function {

    @Override
    public double eval(double x) {
        return x * x;
    }

    @Override
    public List<Double> eval(List<Double> x) {
        List<Double> result = new ArrayList<>();
        for (Double val : x) {
            result.add(this.eval(val));
        }
        return result;
    }

    // no constructors because Square has no fields
}
```

`Square` implements the **`Function`** interface

`Square` must provide an implementation of **`eval(double)`**

`Square` must provide an implementation of **`eval(List<Double>)`**

```java
public class Reciprocal implements Function {

    @Override
    public double eval(double x) {
        return 1.0 / x;
    }

    @Override
    public List<Double> eval(List<Double> x) {
        List<Double> result = new ArrayList<>();
        for (Double val : x) {
            result.add(this.eval(val));
        }
        return result;
    }

    // no constructors because Reciprocal has no fields
}
```

Reciprocal implements the **Function** interface

Reciprocal must provide an implementation of **eval(double)**

Reciprocal must provide an implementation of **eval(List<Double>)**

```
public class SquareWave implements Function {

    private int nmax;

    public SquareWave(int nmax) {
        if (nmax < 1) {
            throw new IllegalArgumentException();
        }
        this.nmax = nmax;
    }

    @Override
    public double eval(double x) {
        double result = 0;
        for (int n = 1; n < this.nmax; n += 2) {
            result += Math.sin(n * Math.PI * x) / n;
        }
        return 4 / Math.PI * result;
    }
```

SquareWave implements the **Function** interface

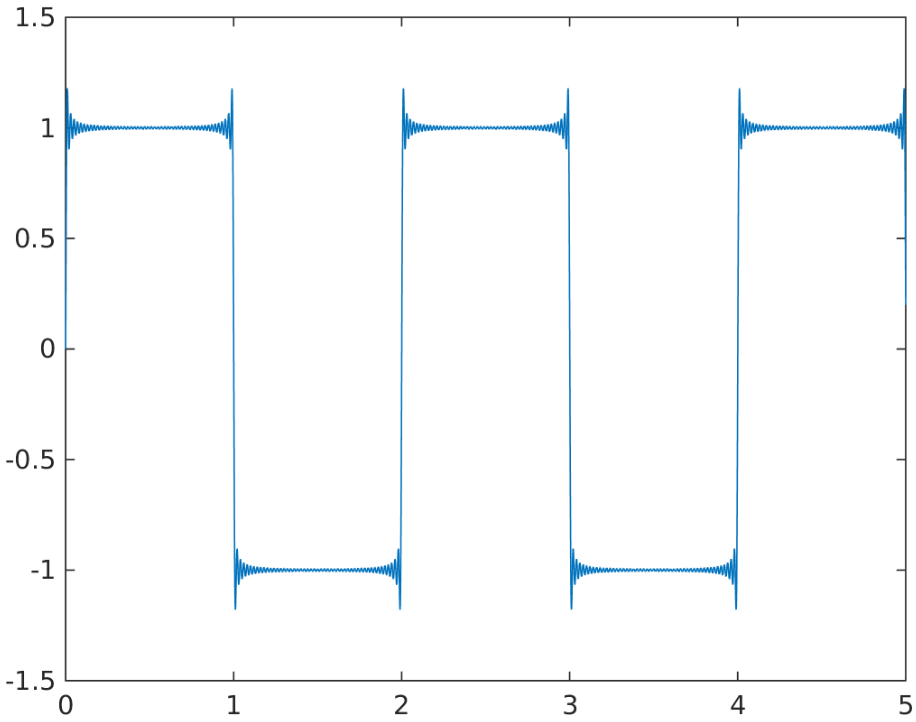
SquareWave must provide an implementation of **eval(double)**

```java
@Override
public List<Double> eval(List<Double> x) {
    List<Double> result = new ArrayList<>();
    for (Double val : x) {
        result.add(this.eval(val));
    }
    return result;
}
```

SquareWave must provide an implementation of **eval(List<Double>)**

SquareWave

- **`SquareWave`** implements the Fourier series for a square wave
  - results for **`nmax = 101`**



Interfaces in the Java library

- interfaces are widely used in the Java library
  - `Collection`, `List`, `Set`, `Map`
  - `Iterable`, `Iterator`
  - `CharSequence` , `Appendable`
  - `Comparable`
  - …

Interfaces are types

- an interface is a reference data type
  - if you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface (https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html)

```
List<String> t = new ArrayList<String>();
```

interface

implements the interface

Interfaces are types

- an interface is a reference data type
  - if you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface (https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html)

```
Function f = new SquareWave(101);
```

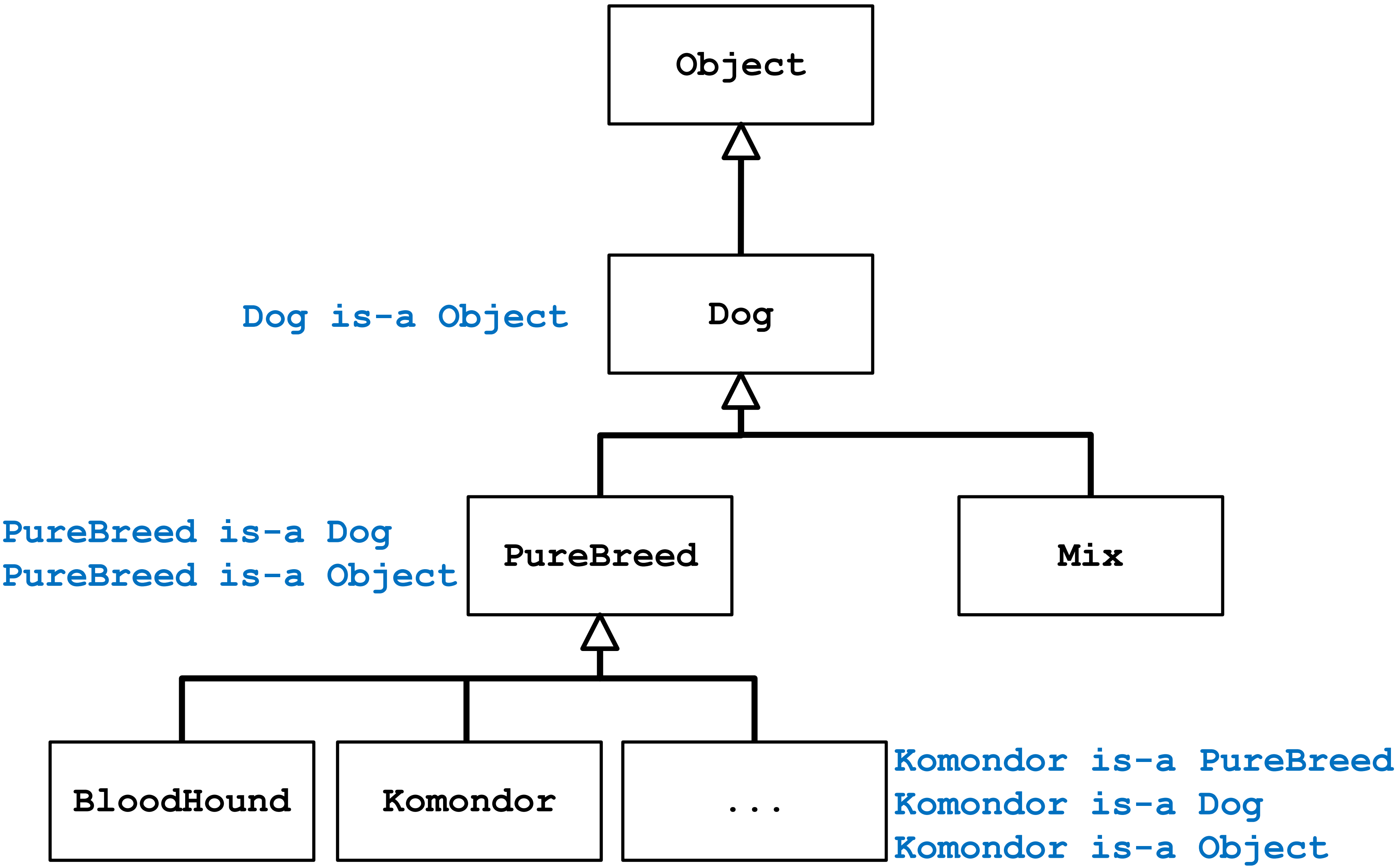
interface — implements the interface

Inheritance

Notes Chapter 6

Inheritance

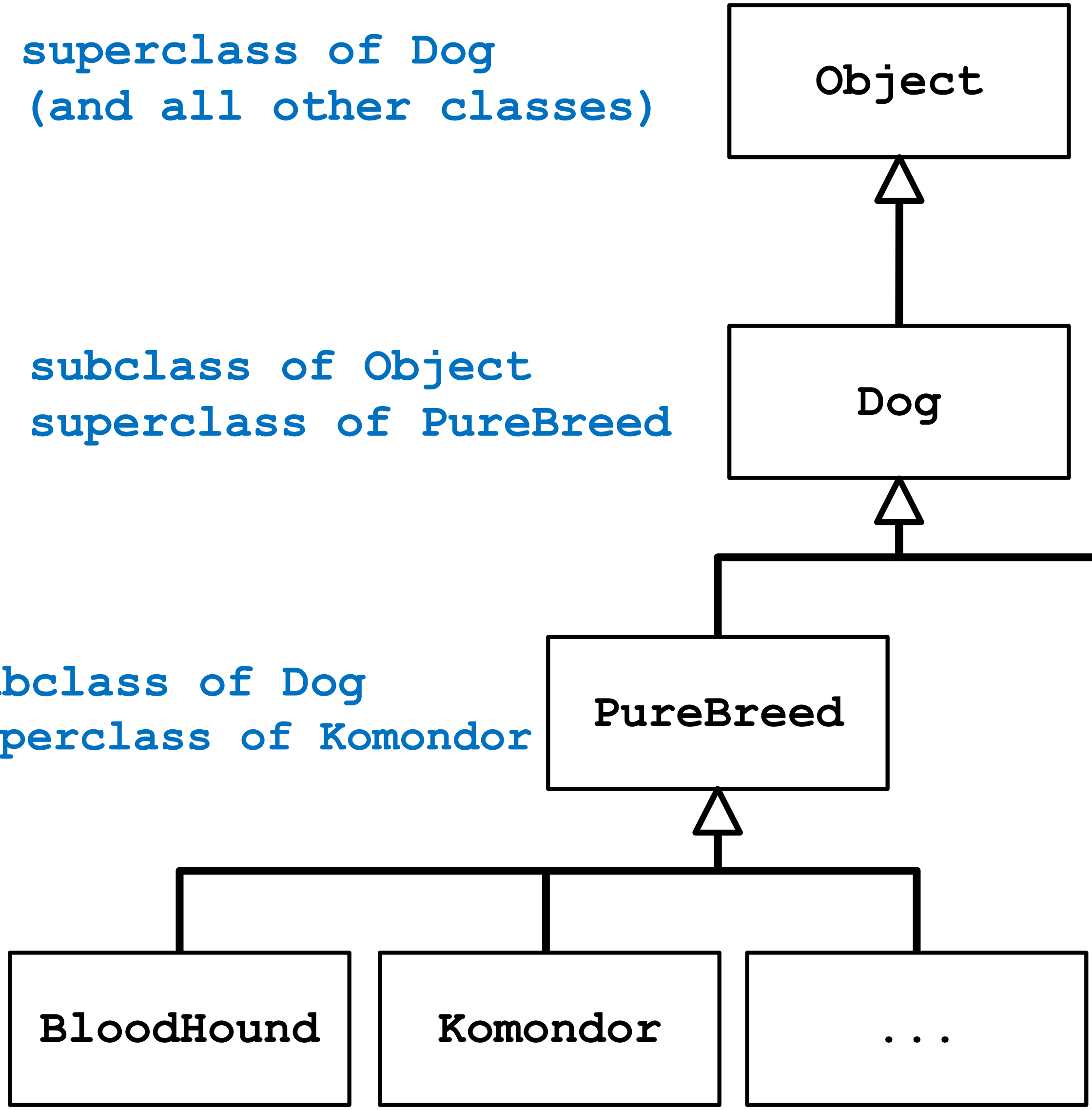
- you know a lot about an object by knowing its class
  - for example what is a Komondor?


Object
Dog
Dog is-a Object
PureBreed
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)

subclass of Object
superclass of PureBreed

subclass of Dog
superclass of Komondor

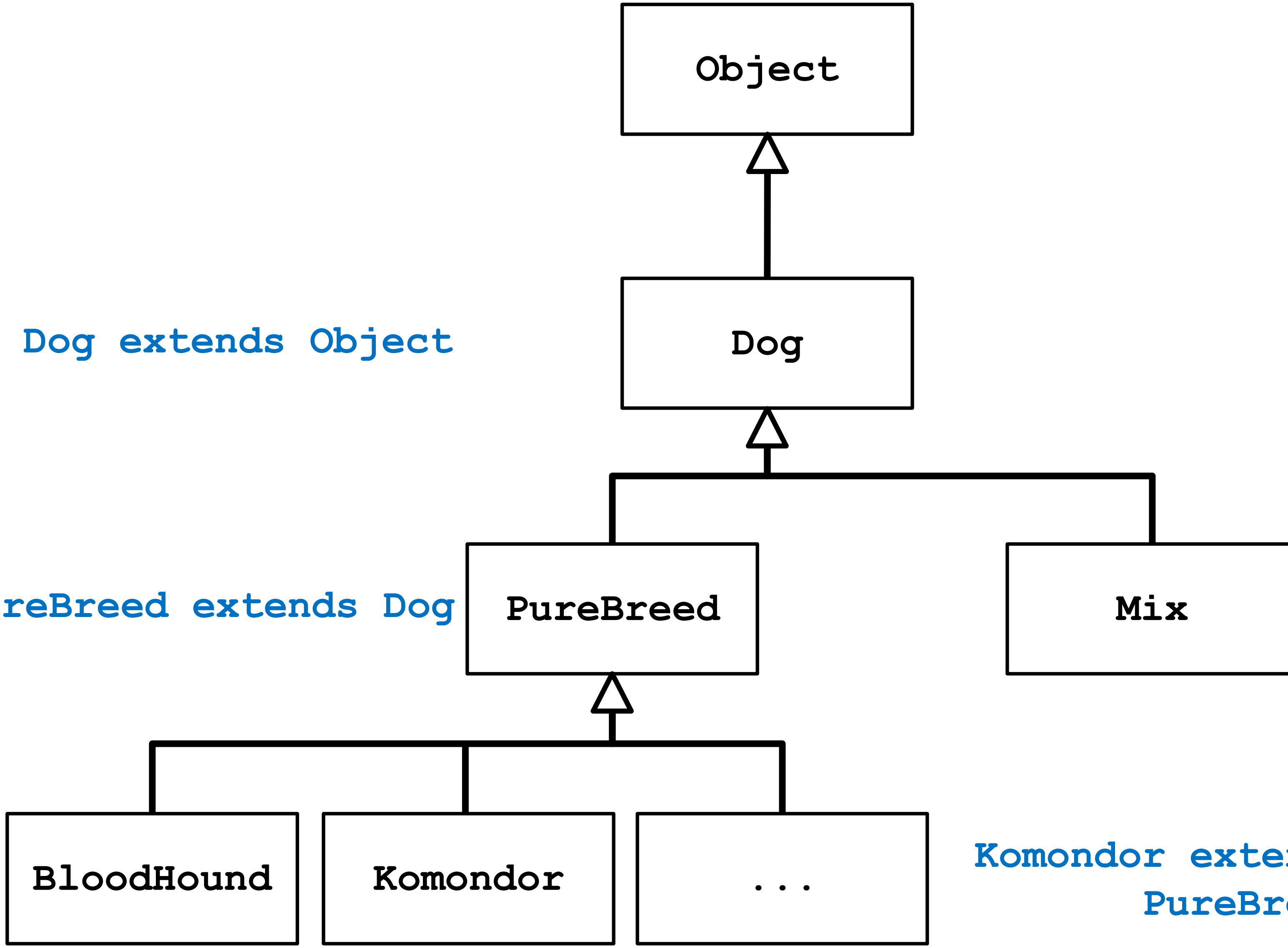


superclass ==
base class
parent class

subclass ==
derived class
extended class
child class

Dog extends Object

PureBreed extends Dog



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 (fields 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
  - *is-a* means *is-substitutable-for*

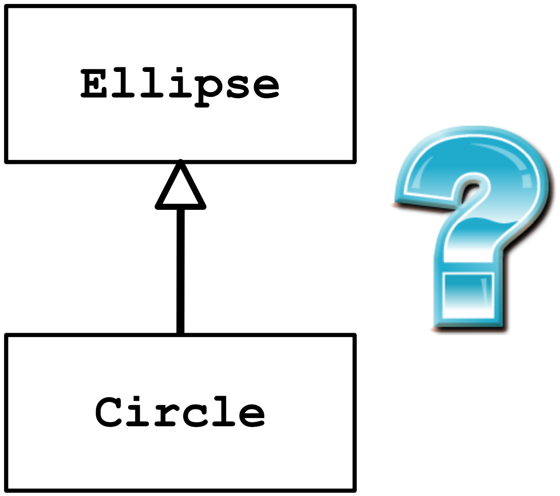
Is-A

- from a Java point of view, is-a means you can use a derived class instance in place of an ancestor class instance

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


// client code of someMethod

Komondor shaggy = new Komondor();
SomeClass.someMethod( shaggy );   // OK, Komondor is-a dog

Mix mutt = new Mix ();
SomeClass.someMethod( mutt );     // OK, Mix is-a dog
```

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`?

- mathematically a circle is a kind of ellipse
- *but* if **`Ellipse`** can do something that **`Circle`** cannot, then **`Circle`** is-a **`Ellipse`** is false for the purposes of inheritance
  - 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 fields **`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

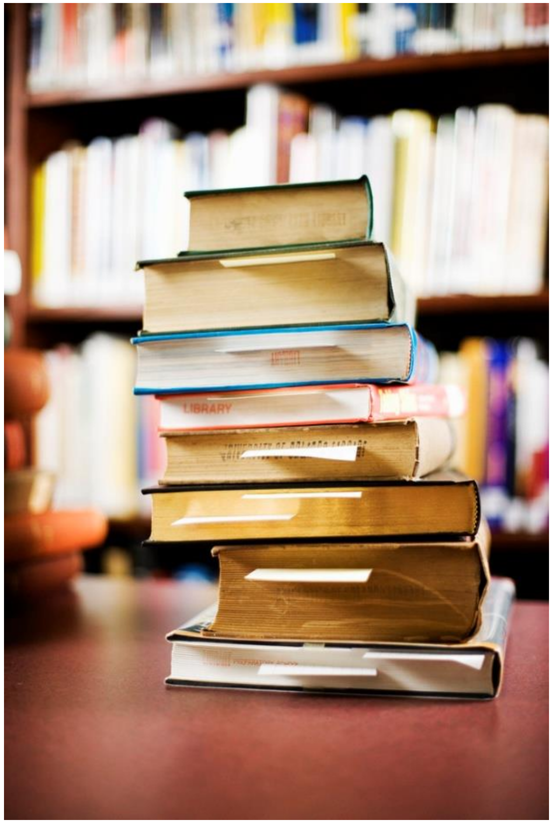
- what if there is no **`setSize`** method?
  - if a **`Circle`** can do everything an **`Ellipse`** can do then **`Circle`** can extend **`Ellipse`**

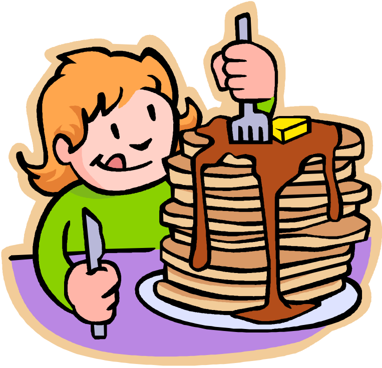
A Naïve Inheritance Example

- a stack is an important data structure in computer science
  - data structure: an organization of information for better algorithm efficiency or conceptual unity
    - e.g., list, set, map, array
- widely used in computer science and computer engineering
  - e.g., undo/redo can be implemented using two stacks

Stack

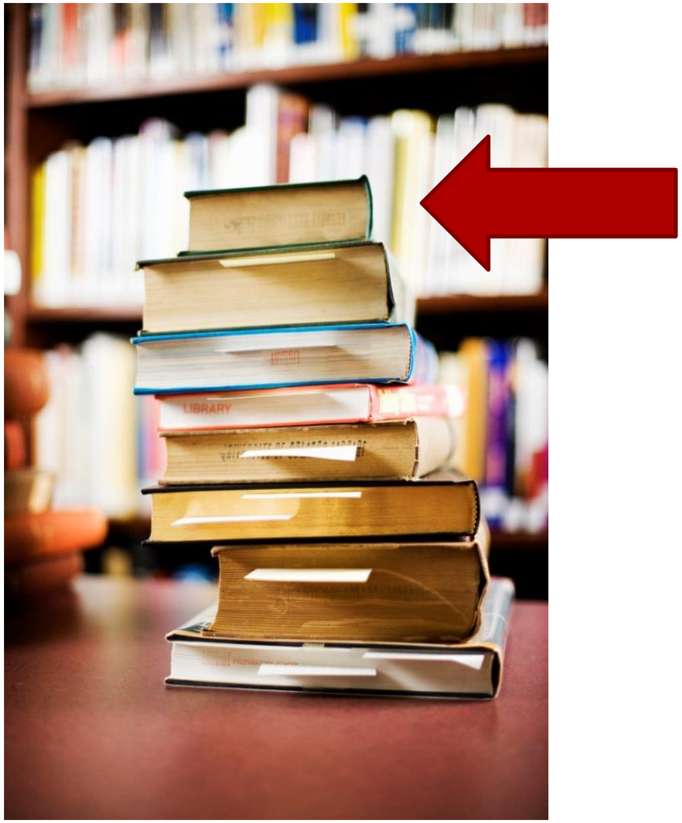
- examples of stacks





Top of Stack

- top of the stack



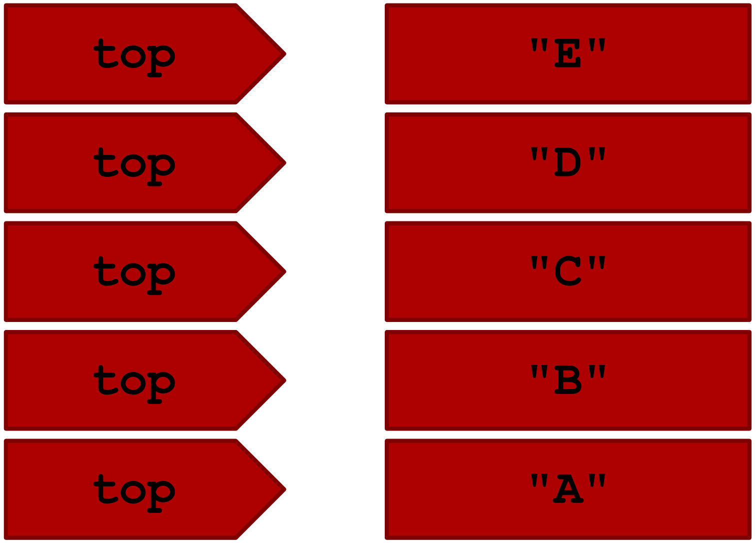
Stack Operations

- classically, stacks only support two operations
  1. push
     - add to the top of the stack
  2. pop
     - remove from the top of the stack

- there is no way to access elements of the stack except at the top of the stack

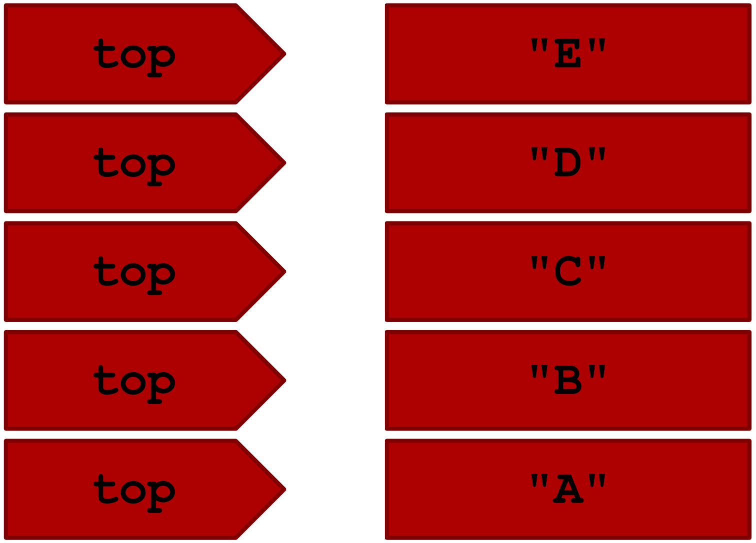
Push

1. `st.push("A")`
2. `st.push("B")`
3. `st.push("C")`
4. `st.push("D")`
5. `st.push("E")`



Pop

```
1.  String s = st.pop()
2.  s = st.pop()
3.  s = st.pop()
4.  s = st.pop()
5.  s = st.pop()
```



Implementing stack using inheritance

- a stack looks a lot like a list
  - pushing an element onto the top of the stack looks like adding an element to the end of a list
  - popping an element from the top of a stack looks like removing an element from the end of the list
- if we have stack inherit from list, our stack class inherits the **`add`** and **`remove`** methods from list
  - we don't have to implement them ourselves
- let's try making a stack of integers by inheriting from **`ArrayList<Integer>`**

Implementing stack using inheritance

```
import java.util.ArrayList;

public class BadStack extends ArrayList<Integer> {


}
```

use the keyword **`extends`** followed by the name of the class that you want to extend

Implementing stack using inheritance

```
import java.util.ArrayList;

public class BadStack extends ArrayList<Integer> {

  public void push(int value) {
    this.add(value);
  }

  public int pop() {
    int last = this.remove(this.size() - 1);
    return last;
  }

}
```

push = add to end of this list

pop = remove from end of this list

Implementing stack using inheritance

- that's it, we're done!

```
public static void main(String[] args) {
  BadStack t = new BadStack();
  t.push(0);
  t.push(1);
  t.push(2);
  System.out.println(t);
  System.out.println("pop: " + t.pop());
  System.out.println("pop: " + t.pop());
  System.out.println("pop: " + t.pop());
}
```

```
[0, 1, 2]
pop: 2
pop: 1
pop: 0
```

Implementing stack using inheritance

- why is this a poor implementation?
- by having **`BadStack`** inherit from **`ArrayList<Integer>`** we are saying that a stack is a list
  - anything a list can do, a stack can also do, such as:
    - get a element from the middle of the stack (instead of only from the top of the stack)
    - set an element in the middle of the stack
    - iterate over the elements of the stack

Implementing stack using inheritance

```java
public static void main(String[] args) {
   BadStack t = new BadStack();
   t.push(100);
   t.push(200);
   t.push(300);
   System.out.println("get(1)?: " + t.get(1));
   t.set(1, -1000);
   System.out.println("set(1, -1000)?: " + t);
 }
```

```
[100, 200, 300]
get(1)?: 200
set(1, -1000)?: [100, -1000, 300]
```

Implementing stack using inheritance

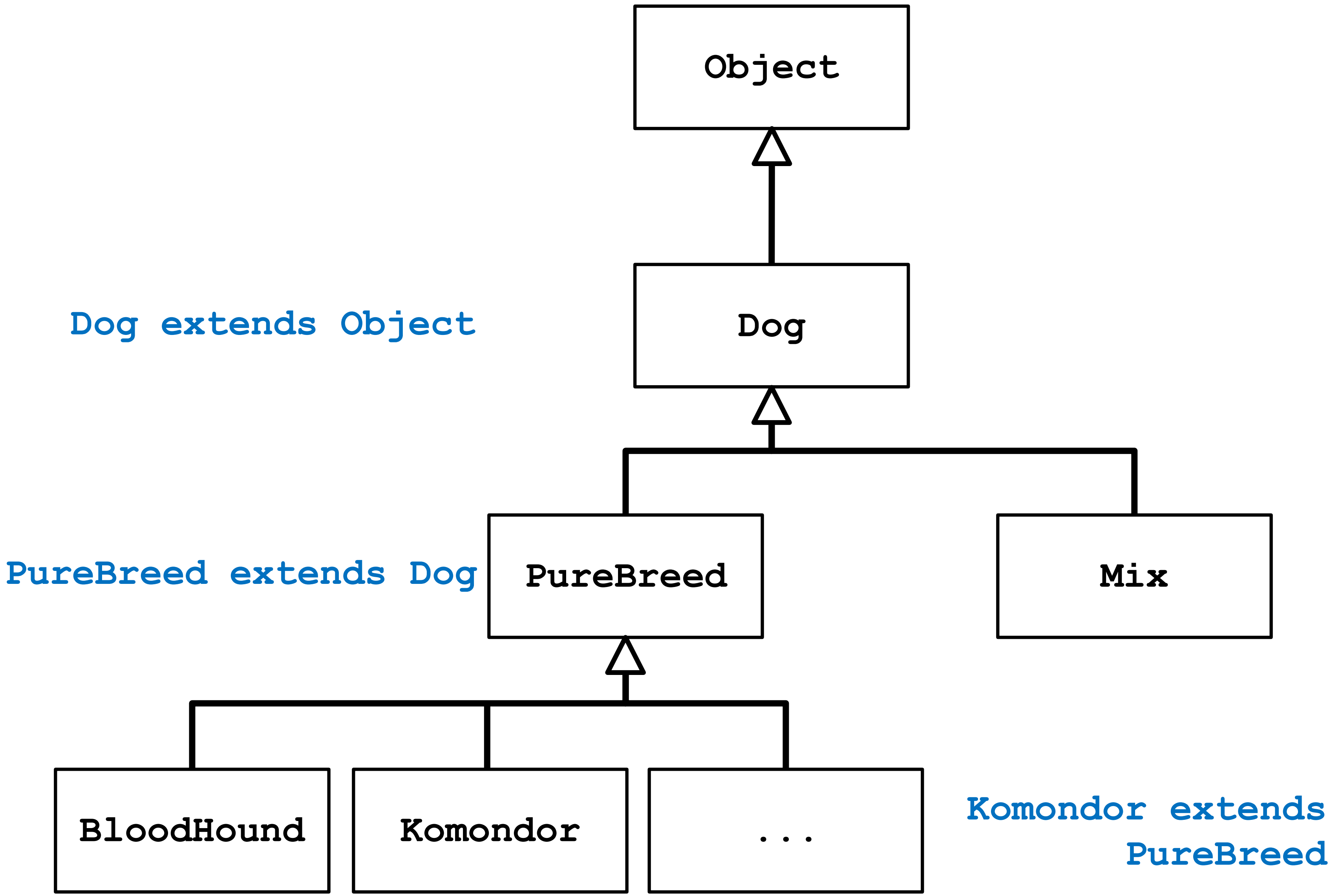
- using inheritance to implement a stack is an example of an incorrect usage of inheritance
- inheritance should only be used when an is-a relationship exists
  - a stack is not a list, therefore, we should not use inheritance to implement a stack
- even experts sometimes get this wrong
  - early versions of the Java class library provided a stack class that inherited from a list-like class
    - `java.util.Stack`

Other ways to implement stack

- use composition
- **`Stack`** has-a **`List`**
- the end of the list is the top of the stack
  - **`push`** adds an element to the end of the list
  - **`pop`** removes the element at the end of the list

Inheritance (Part 2)

Notes Chapter 6


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

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...

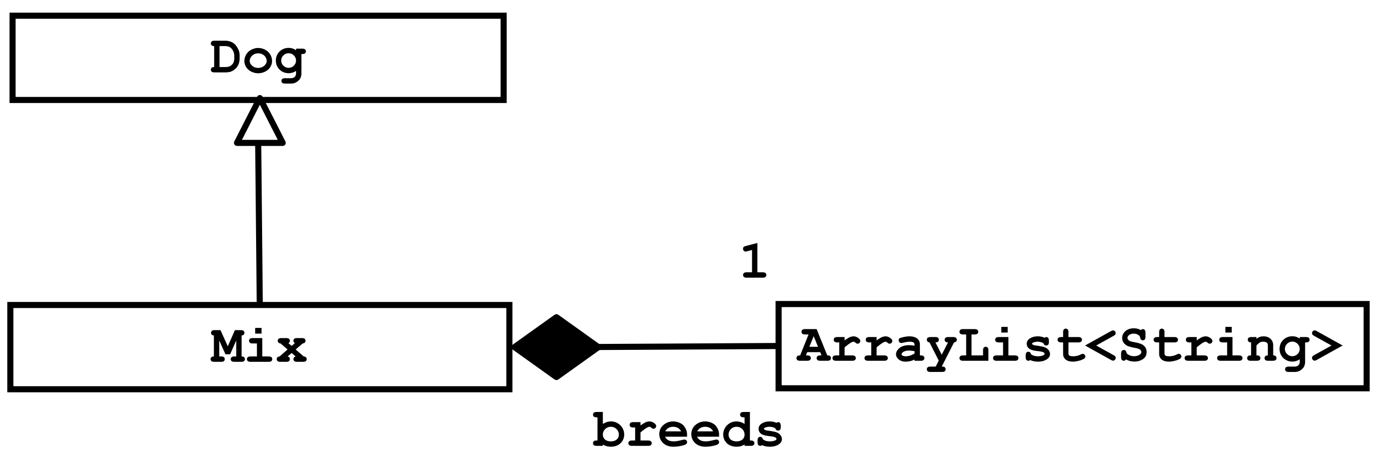
What is a Subclass?

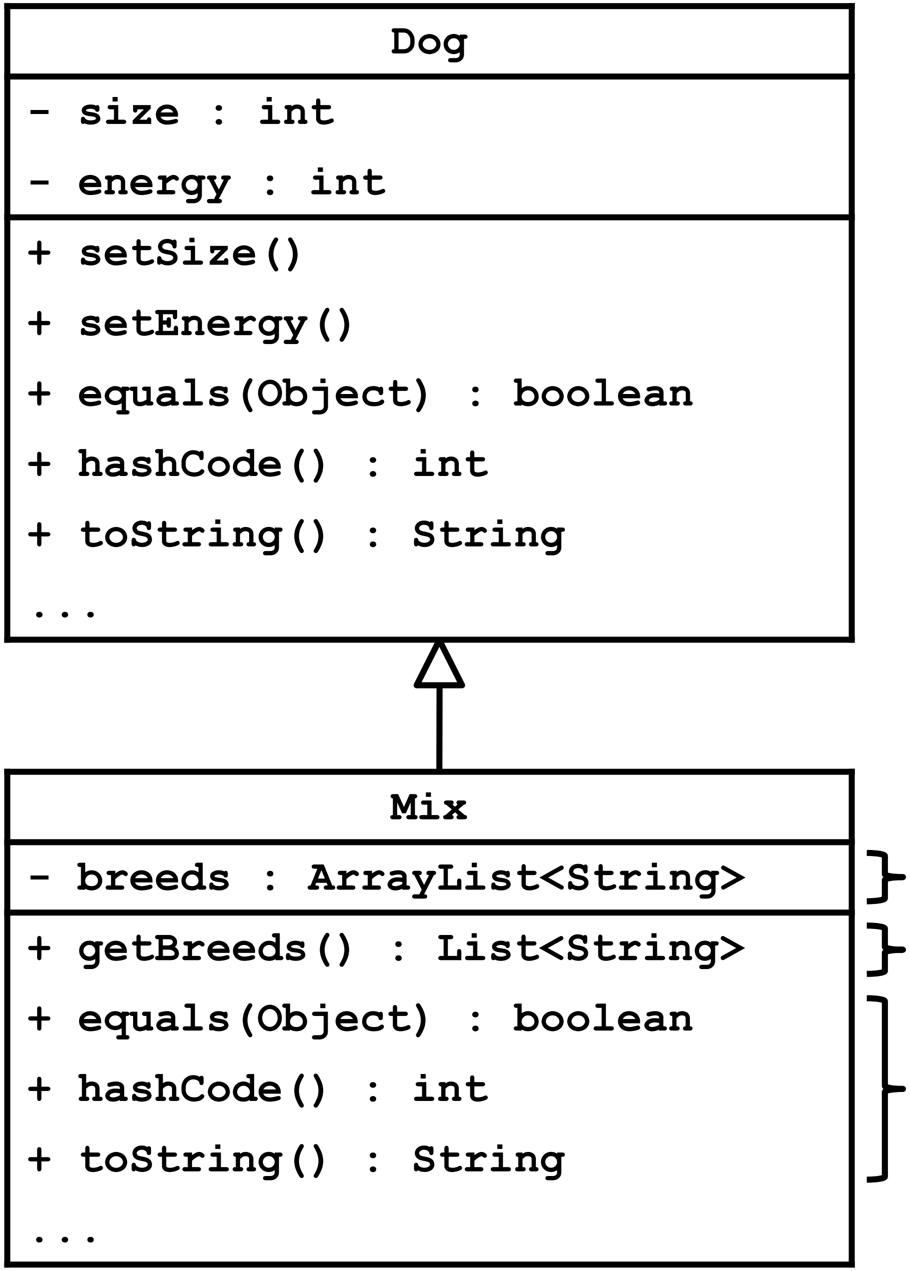
- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
  - the new class has direct access to the **`public`** and **`protected`*** fields and methods without having to re-declare or re-implement them
  - the new class can introduce new fields and methods
  - the new class can re-define (override) its superclass methods

* the notes does not discuss **`protected`** access

Mix UML Diagram

- a mixed breed dog is a dog whose ancestry is unknown or includes more than one pure breed





- subclass can add new fields
- subclass can add new methods
- subclass can change the implementation of inherited methods

What is a Subclass?

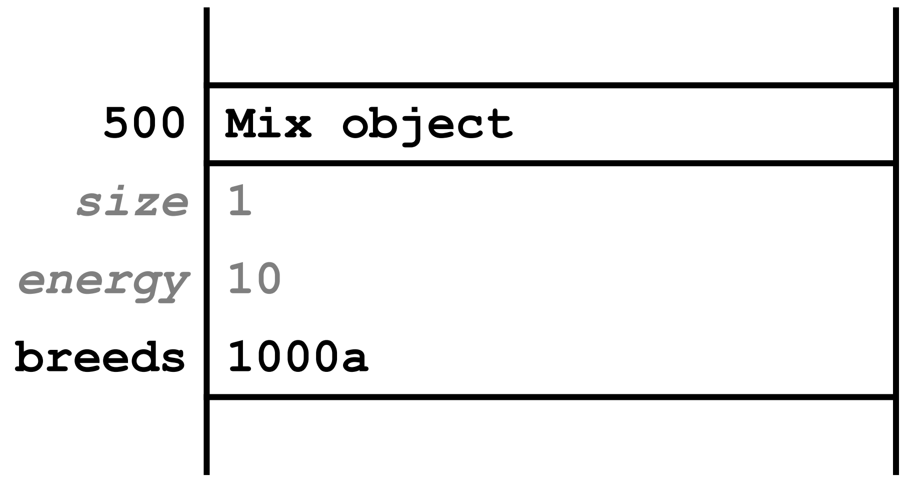
- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
- inheritance does more than copy the API of the superclass
  - the derived class contains a subobject of the parent class
  - the superclass subobject needs to be constructed (just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

What is a Subclass?

- another model of inheritance is to imagine that the subclass contains all of the fields of the parent class (including the private fields), but cannot directly use the private fields

Mix Memory Diagram

- ***size*** and ***energy*** belong to the superclass
- private in superclass
- not accessible by name to **`Mix`**



Constructors of Subclasses

- the purpose of a constructor is to set the values of the fields of **`this`** object
- how can a constructor set the value of a field that belongs to the superclass and is **`private`**?
  - by calling the superclass constructor and passing **`this`** as an implicit argument

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; part of 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>();
  }
```

```java
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 using chaining)

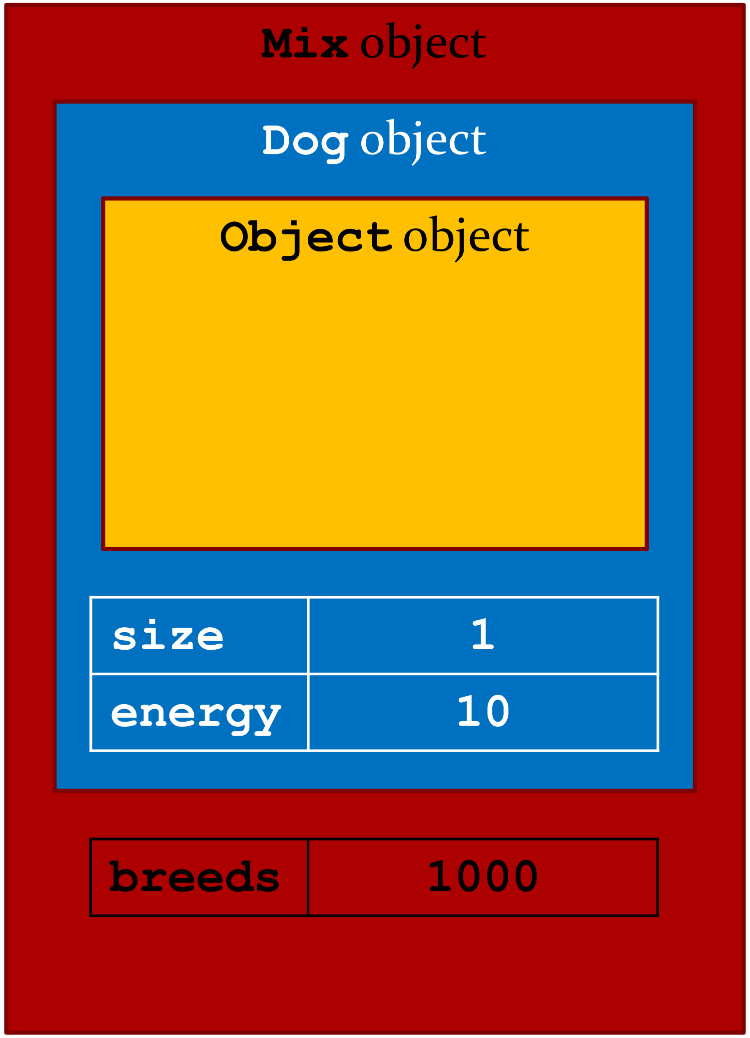
```
public final class Mix extends Dog {
  // no declaration of size or energy; part of Dog
  private ArrayList<String> breeds;

  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>());
  }
```

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

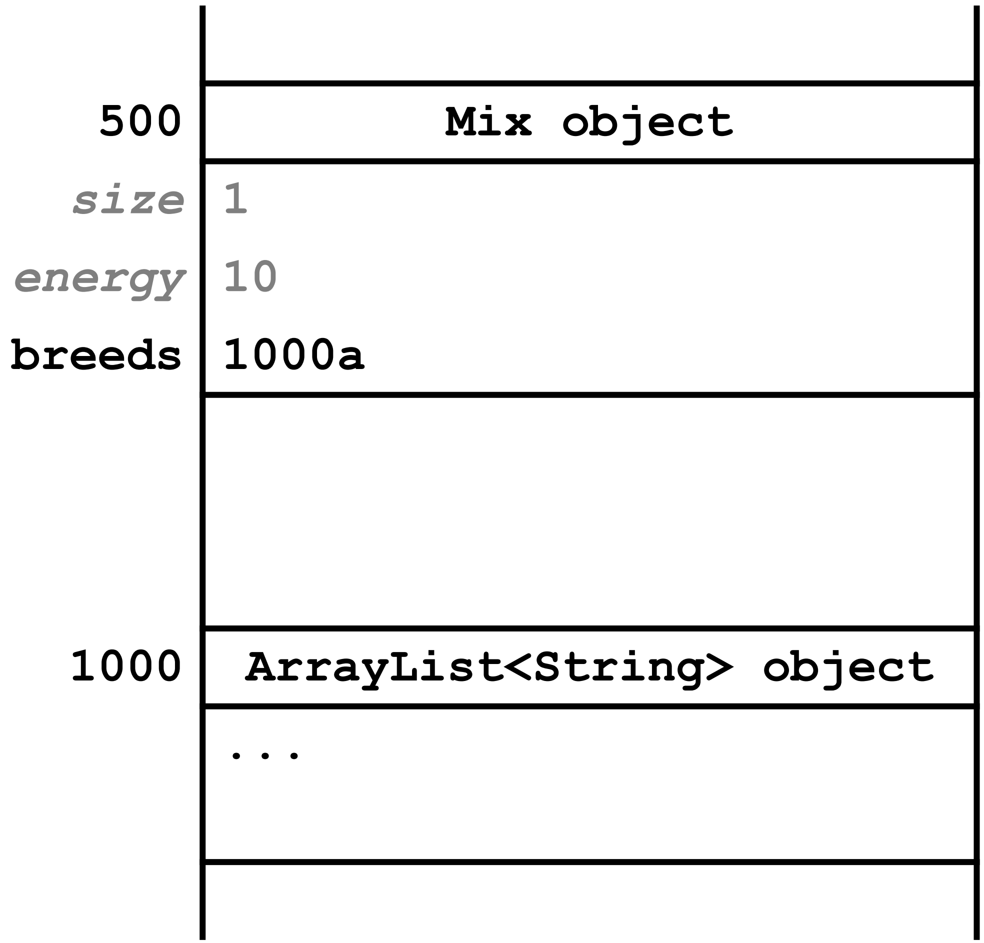
- why is the constructor call to the superclass needed?
  - because **`Mix`** is-a **`Dog`** and the **`Dog`** part of **`Mix`** needs to be constructed

`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
        - and finishes
    - sets `size` and `energy`
    - and finishes
- creates a new empty `ArrayList` and assigns it to `breeds`
- and finishes



Mix Memory Diagram



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

```java
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.println("SubbyDubby overrideMe : " + 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`** field 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

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?

Preconditions and Inheritance

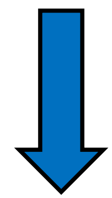
- a subclass can change a precondition on a method but whatever argument values the superclass method accepts must also be accepted by the subclass method

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
// 4. energy == 5
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?

Postconditions and Inheritance

- a subclass can change a postcondition on a method but whatever the superclass method promises will be true when it returns must also be true when the subclass method returns

Strength of a Postcondition

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

```
// Dog getSize
// 1. no postcondition
// 2. return value >= 1
// 3. return value
//        between 1 and 10
// 4. return 5
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

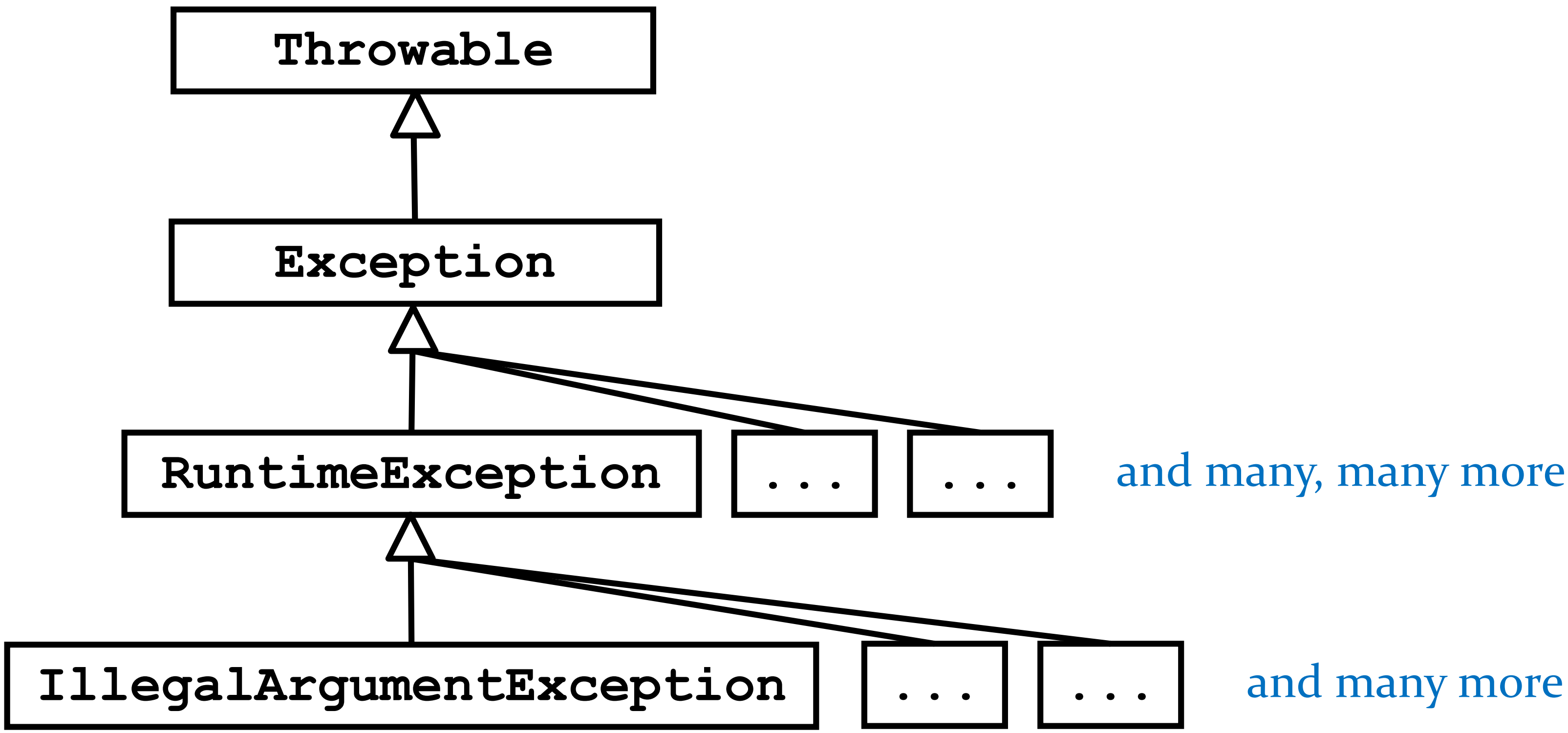
- client code written for **`Dog`**s can now fail when given a **`Dogzilla`**

```
// 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

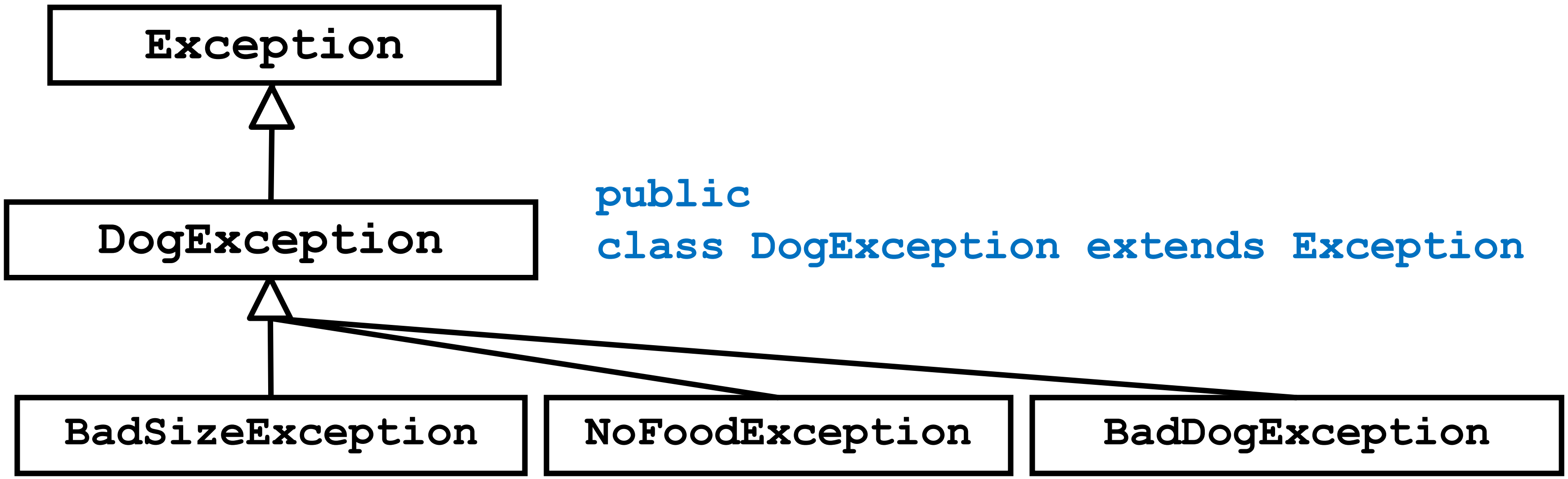
Exceptions

- 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 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
{
  // ...
}
```