EECS 2030
Advanced Object-Oriented Programming

S2023, Section A

Inheritance

Review

Is-A Pitfalls

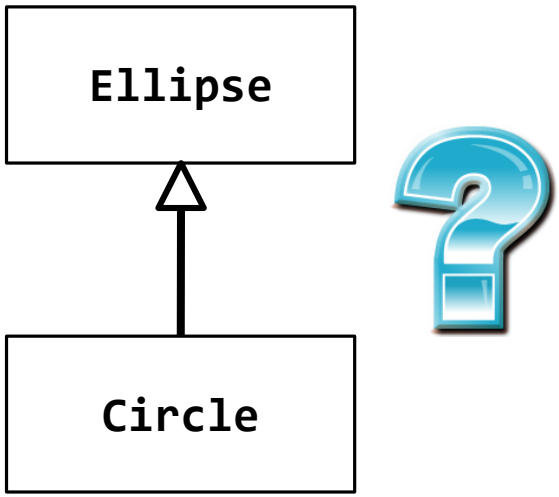
is-a has nothing to do with the real world

is-a has everything to do with how the implementer has modelled the inheritance hierarchy

the classic example:

`Circle` is-a **`Ellipse`**?

In English: “Circle is just a *special kind* of an ellipse”



Implementing stack using composition

```
    /**
     * Pushes a value onto the top of the stack.
     *
     * @param value the value to push onto the stack
     */
    public void push(int value) {
        this.elems.add(value);
    }

    /**
     * Pops a value from the top of the stack.
     *
     * @return the value that was popped from the stack
     */
    public int pop() {
        int last = this.elems.remove(this.elems.size() - 1);
        return last;
    }
}
```

Inheritance for code reuse

`protected` access modifier allows subclasses to access a field defined in a superclass

We should also **modify the contract** of the advance method to indicate that *subclasses might change the behavior of the method*

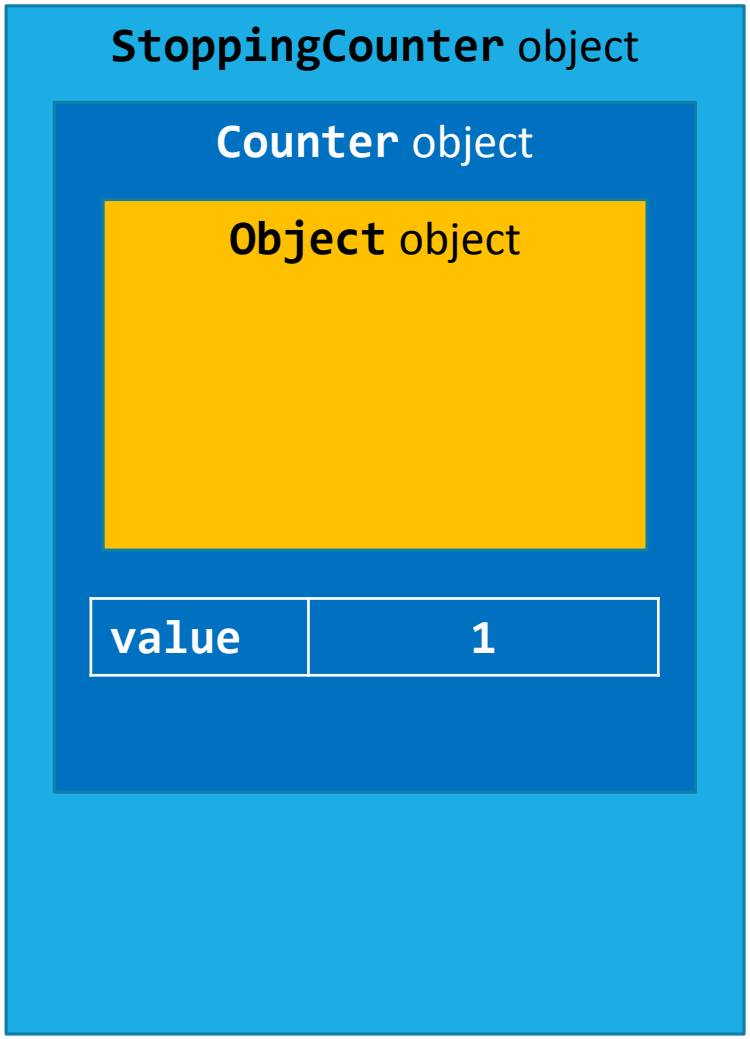
A call to another constructor can only occur on the first line in the body of a **constructor**

A class that throws an **exception** while the superclass didn't is not a good idea (e.g., when counter value reaches **`Integer.MAX_VALUE`**)

```
/**
 * Increment the value of this counter upwards by 1. If this
 * method is called when the current value of this counter is
 * equal to {@code Integer.MAX_VALUE} then the value of this
 * counter is set to 0 (i.e., the counter wraps around to 0)
 * but subclasses can override this behaviour.
 */
public void advance() {
  if (this.value != Integer.MAX_VALUE) {
    this.value++;
  }
  else {
    this.value = 0;
  }
}
```

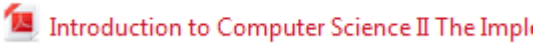
```
StoppingCounter c =
    new StoppingCounter(1);
```

1. **StoppingCounter** constructor starts running
- initializes new **Counter** subobject by invoking the **Counter** constructor
    2. **Counter** constructor starts running
    - initializes new **Object** subobject by (silently) invoking the **Object** constructor
        3. **Object** constructor runs
        - and finishes
    - sets **value**
    - and finishes
- finishes



Inheritance (Part 3)

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

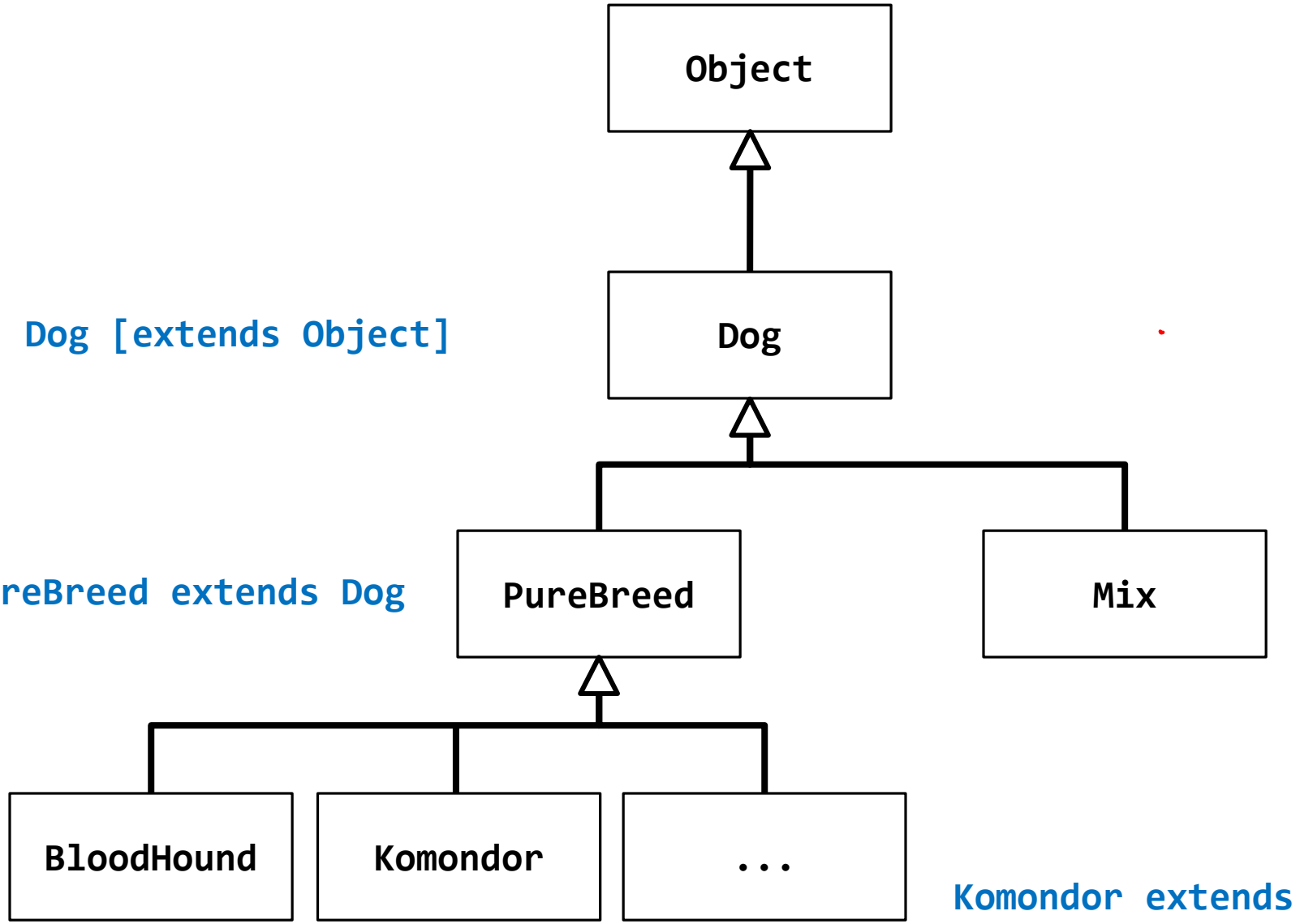
Dog class

Suppose that you want to implement a set of classes to represent **different types of dogs**

E.g., an app that helps pick a suitable dog to adopt

What **traits** might be important for representing dogs?

- breed
- size
- energy
- appearance
- long hair or short hair
- etc.



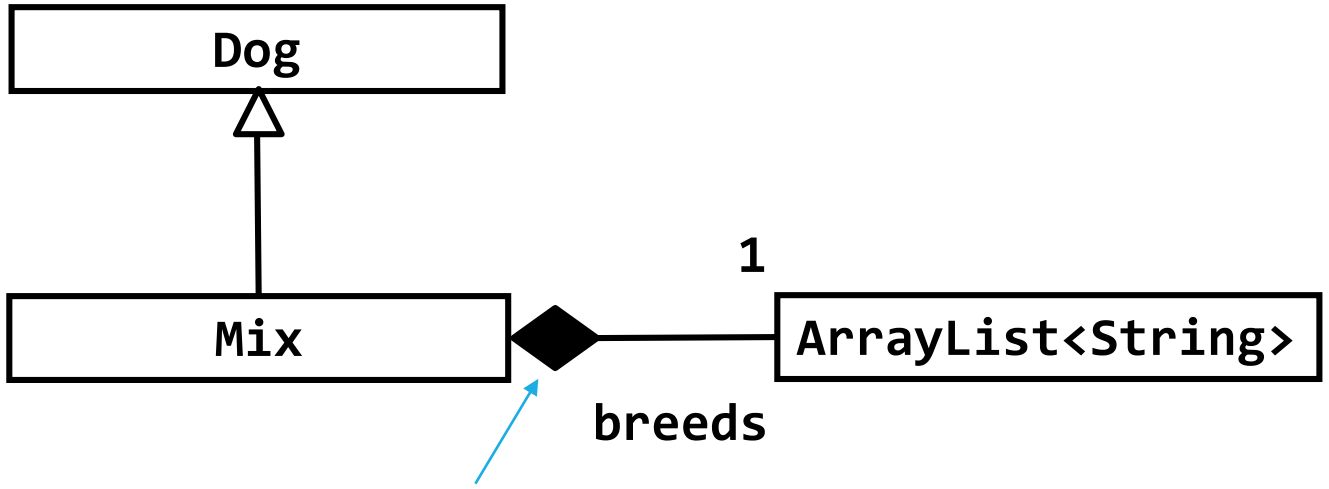
Dog [extends Object]

PureBreed extends Dog

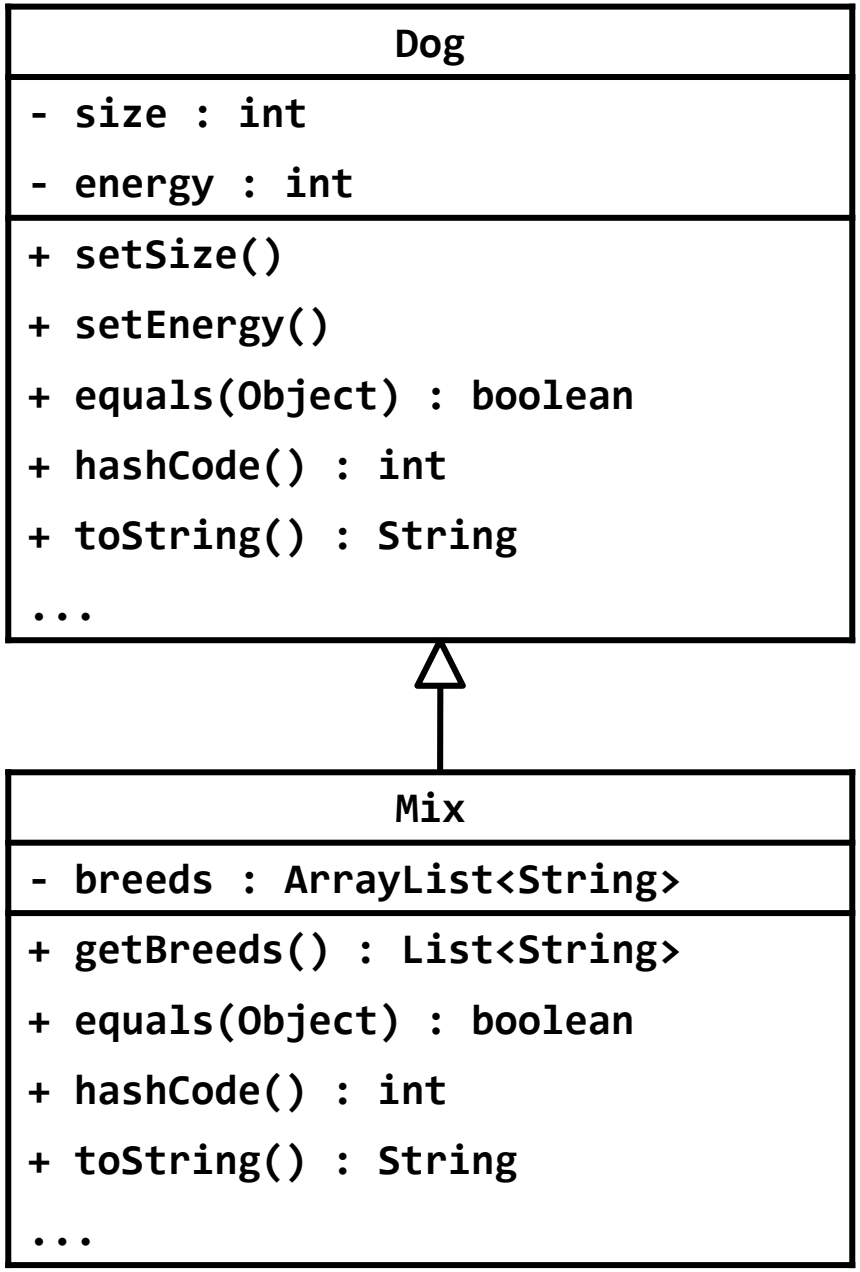
Komondor extends PureBreed

Mix UML Diagram

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



Composition or Aggregation? Why?



- private (-) field named size of type int
- private field named energy of type int
- public (+) method named setSize

- public method that returns a boolean

Other visibility symbols
protected
~ package

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

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

Dog class

our class will have two **invariants**

- **size** is an **int** between 0 and 10 (inclusive)
- **energy** is an **int** between 0 and 10 (inclusive)

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);  }
```

Why method calls?

```
  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…

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 overridable 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();
  }
}
```

Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

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 field date is 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

Method calls from constructor

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

More on this problem: https://en.wikipedia.org/wiki/Fragile_base_class

Preconditions and Inheritance

precondition

a condition that must be true immediately before a method is called

often the precondition involves the arguments passed to the method

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 the modified precondition must be substitutable for the superclass precondition

- e.g., 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 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

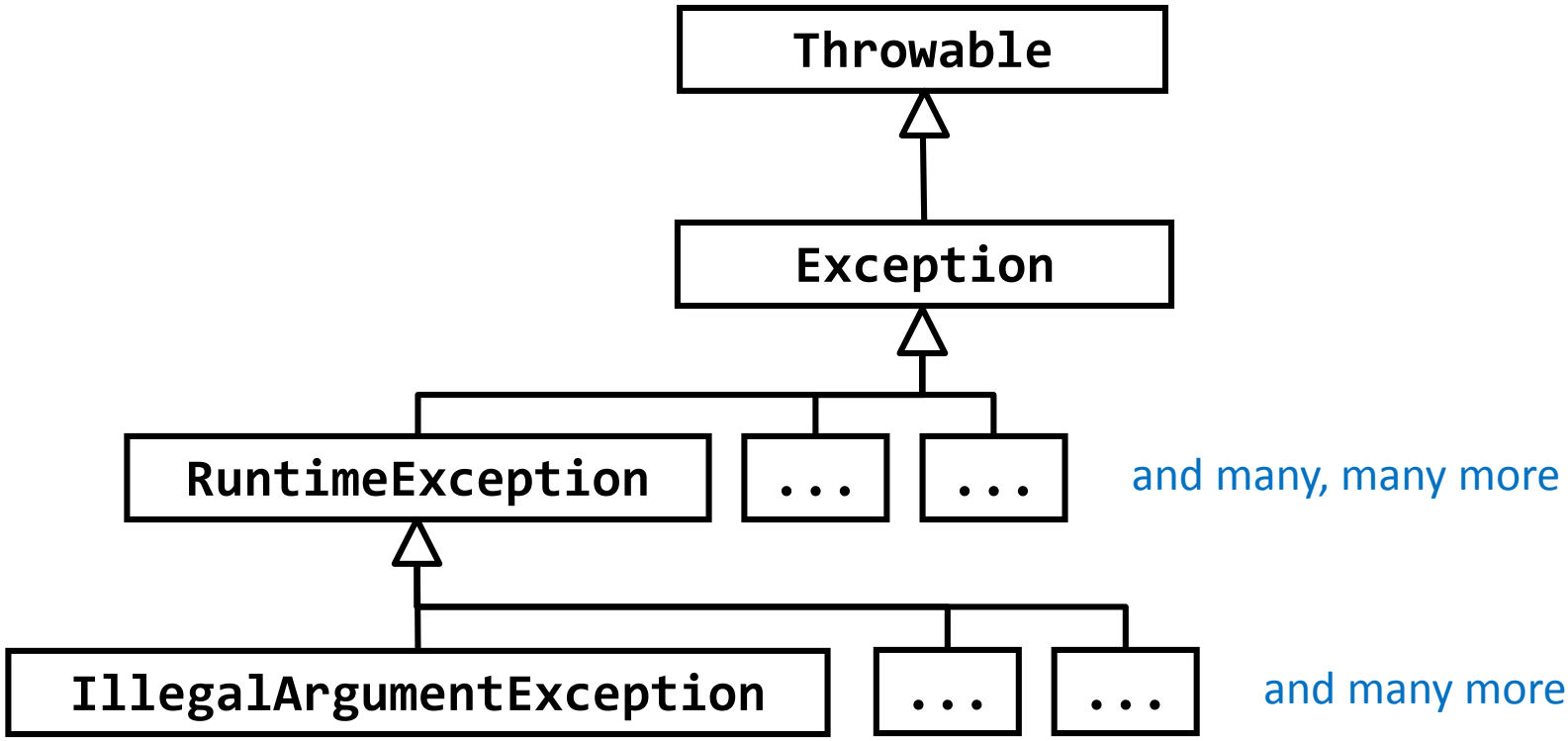
client code for **Dog**s can now fail when given a **Dogzilla**

```
// 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

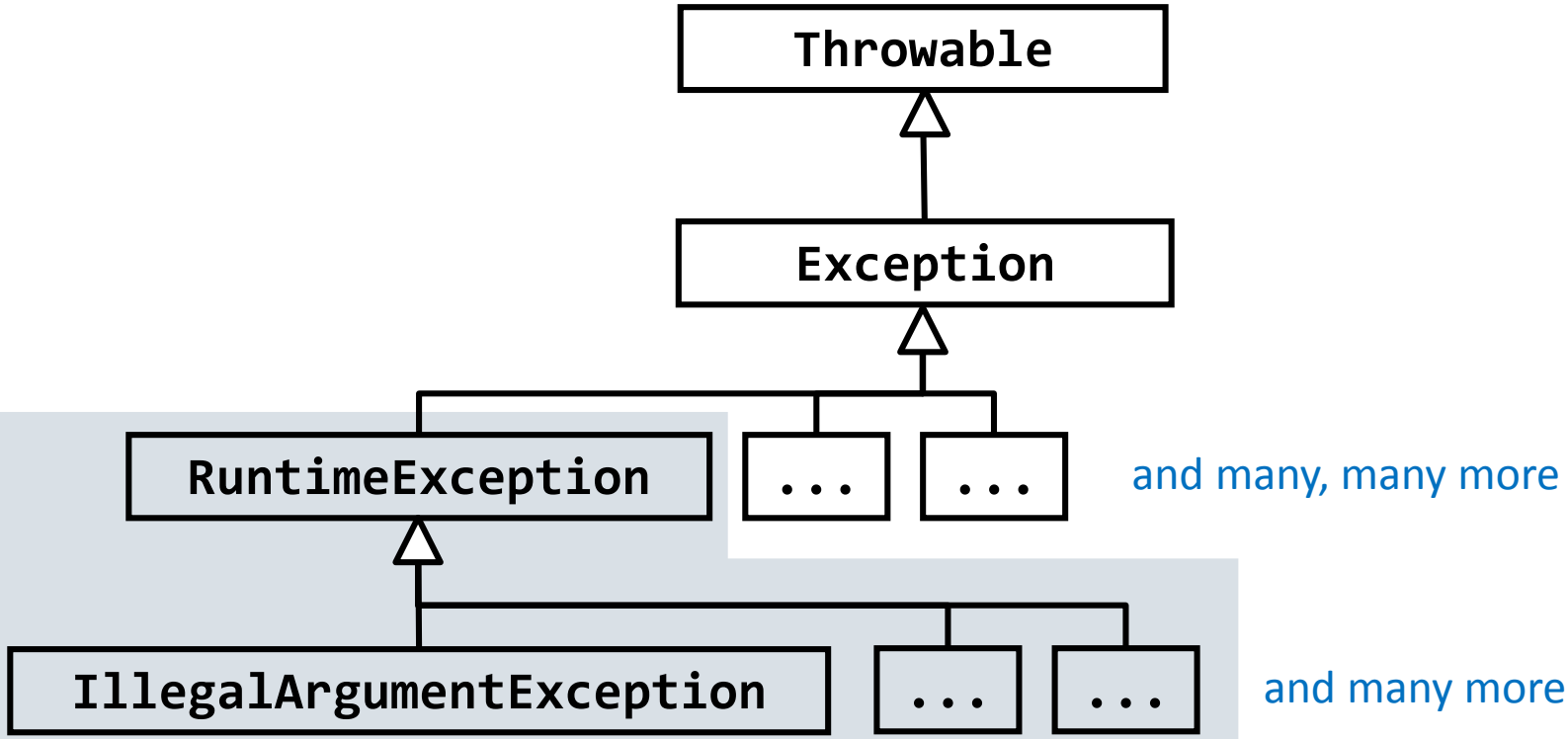
Exceptions

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



Unchecked Exceptions

`RuntimeException` and all of its descendents are called *unchecked exceptions*



Unchecked Exceptions

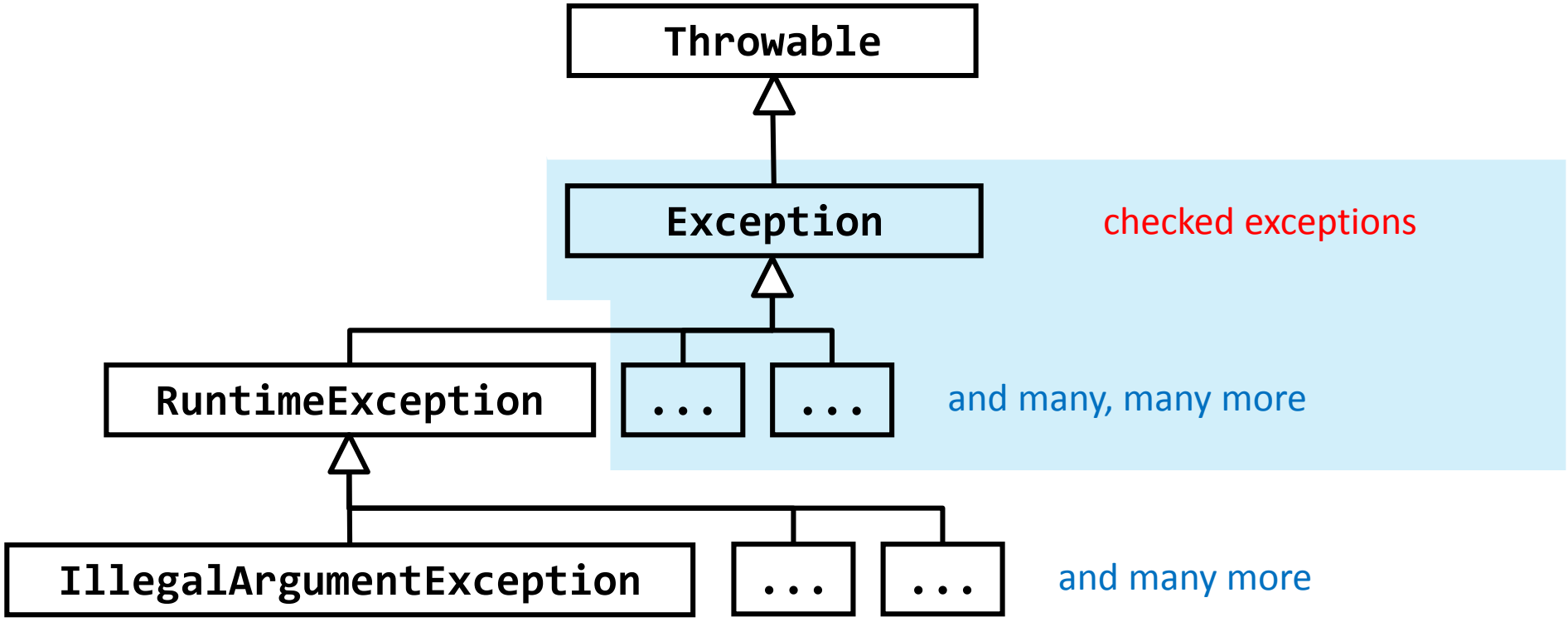
a method may throw an *unchecked* exception *without declaring* that it might throw an exception

- e.g., the following method will throw an **`IndexOutOfBounds`** exception if the list **`dogs`** is empty

```
public void doSomething(List<Dog> dogs) {
    Dog d = dogs.get(0);
    // do something here
}
```

Checked Exceptions

`Exception` and all of its descendants not including the unchecked exceptions are called *checked exceptions*



Checked Exceptions

a method that throws a checked exception must declare that it throws an exception

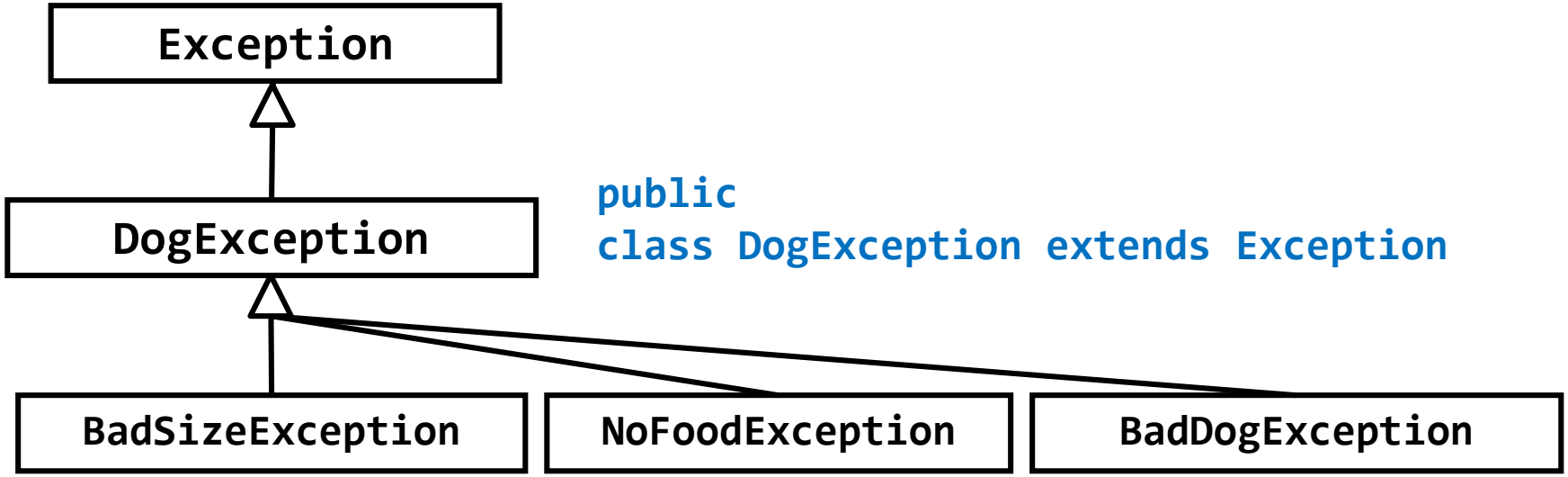
- use the keyword **`throws`** followed by the exception type after the parameter list of the method

```
// in Dog
public void someDogMethod() throws DogException
{
  // some code here
}
```

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 //same class
{
  // ...
}
```

Polymorphism

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() );
}

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

```
Dog toString
CockerSpaniel toString
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() );
}
```

```
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);
```

Before...

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

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! Even though the object itself is of that type…

Dynamic dispatch

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`

selecting which version of a polymorphic method to use at run-time is called *dynamic dispatch*

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**

Another Example

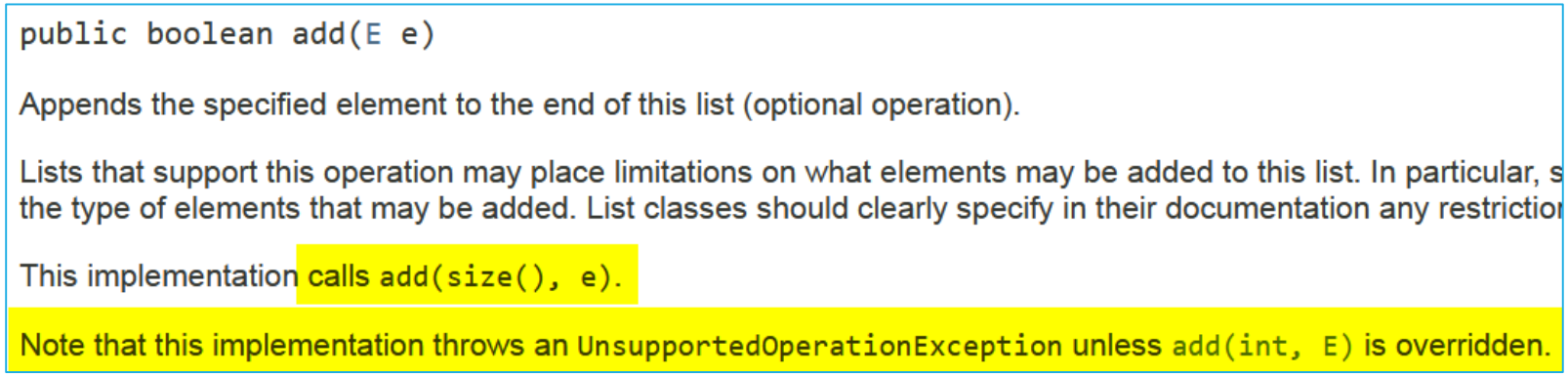
e.g. you might want to know the breed of a **Dog** but only the subclasses have information about the breed

- you want to add the method `getBreed` to **Dog** but only the subclasses of **Dog** can implement `getBreed`

Another Example

Sometimes you just want to have a default implementation of some desired methods, to make implementing subclasses easier

- E.g., AbstractList has some default implementations of the List interface



`public boolean add(E e)`

Appends the specified element to the end of this list (optional operation).

Lists that support this operation may place limitations on what elements may be added to this list. In particular, s
the type of elements that may be added. List classes should clearly specify in their documentation any restrictio

This implementation calls `add(size(), e)`.

Note that this implementation throws an `UnsupportedOperationException` unless `add(int, E)` is overridden.

What is an Abstract Class?

If the base class has methods that only subclasses can define *and* fields common to all subclasses then the base class should be **abstract**

- if you have a base class that has only 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 (the constraints above are not enforced). Just placing a word abstract in front is sufficient!

- e.g. http://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html

How Abstract Classes are Used

an abstract class provides a *partial definition* of a class

- everything that is common to all of the subclasses
- 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 and wishes to leave those 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();
}
```

get

`public abstract E get(int index)`

Returns the element at the specified position in this list.

Specified by:

`get` in interface `List<E>`

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();
}
```

PureBred

A purebred dog is a dog with a single breed
- one `String` field to store the breed

The breed is determined by the subclasses!
- **PureBred** still cannot give the **breed** field a value
- However, it can implement the method **getBreed**

PureBred defines a field common to all subclasses and it needs the subclass to inform it of the actual breed

Thus, **PureBred** is also an abstract class

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

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

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

Note: no abstract methods in this abstract class!

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

}
```

Subclasses of PureBred

the subclasses of **`PureBred`** are responsible for setting the breed

consider **`Komondor`**

Komondor

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

  public Komondor() {
    super(BREED);
  }

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

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

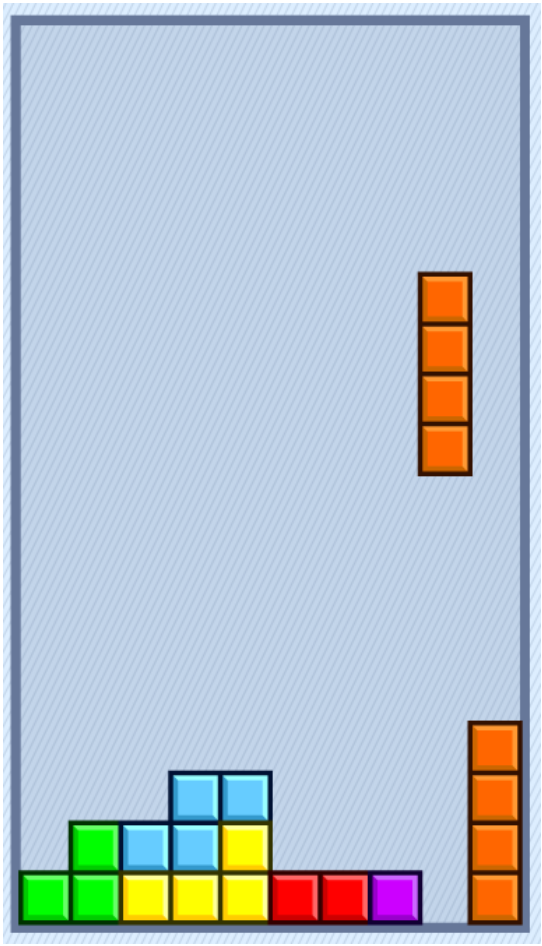
Another example: Tetris

played with 7 standard blocks called tetriminoes

blocks drop from the top

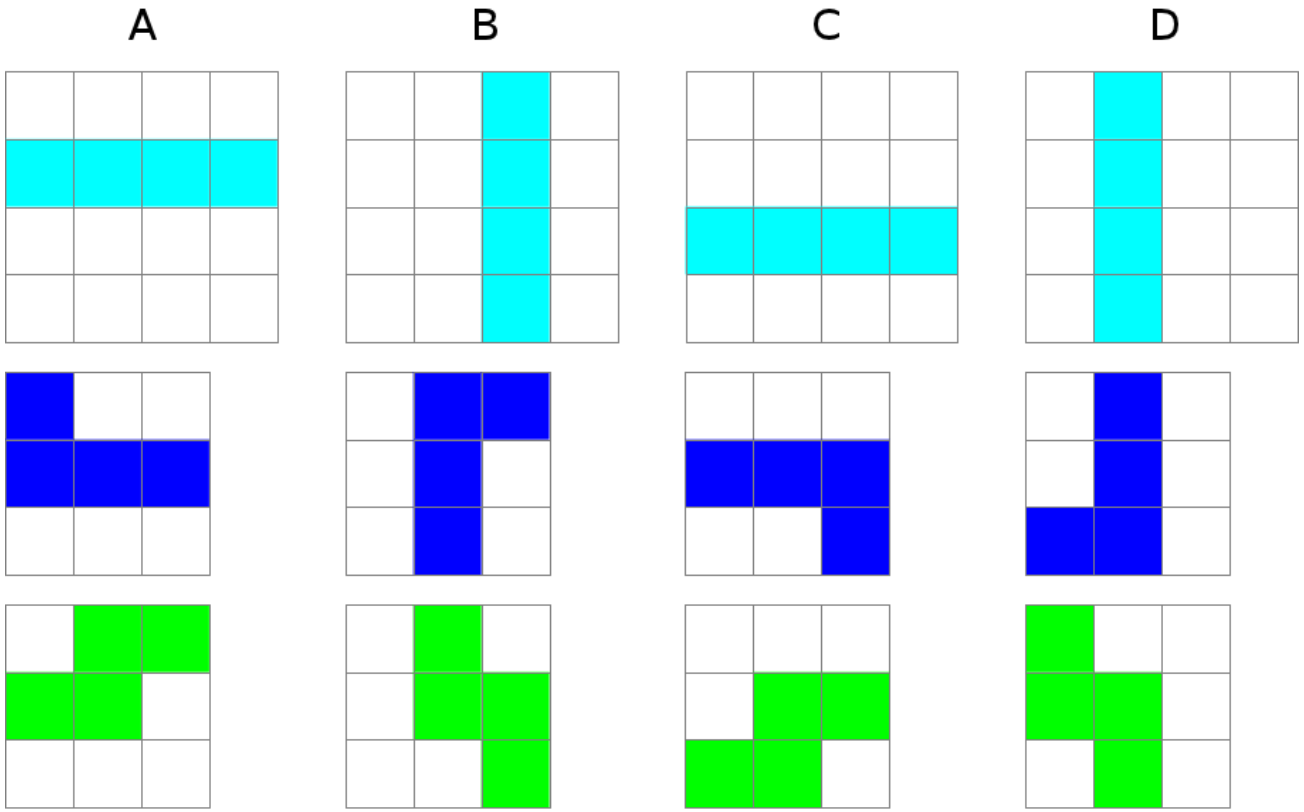
player can move blocks left, right, and down

player can spin blocks left and right



Tetriminoes

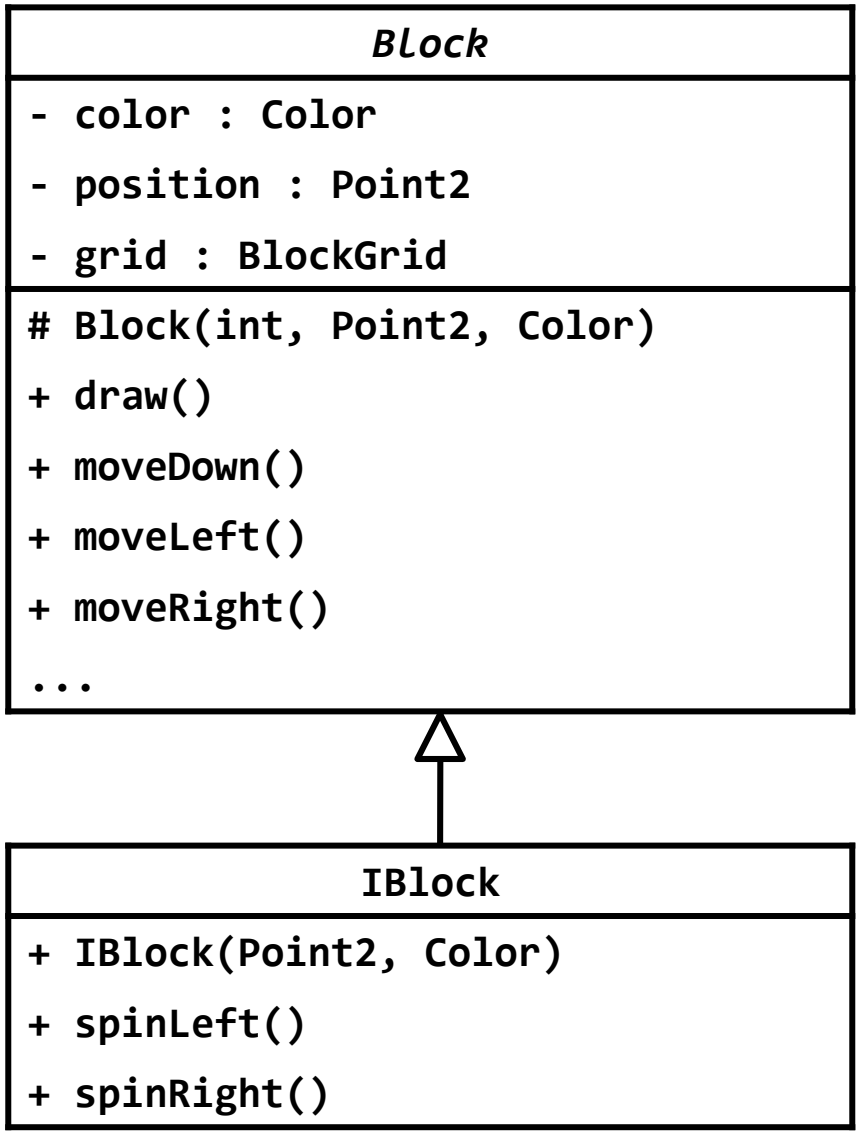
spinning the I, J, and S blocks



Tetriminoes

features **common** to all tetriminoes
- has-a color
- has-a shape
- has-a position
- draw
- move left, right, and down

features **unique** to each kind of tetrimino
- the actual shape
- spin left and right



- class name in italics for abstract classes
- a 2D point
- a grid object that stores the shape
- protected constructor

Block is abstract because we can't define the shape of a generic block

- constructor defines the shape
- methods modify the shape to produce the rotated version of the block

Another example: Counters

Basic counter class counts upwards starting from zero

Other ways a counter can count?

- downwards from some value
- in either direction
- up to some maximum value then back down to some minimum value then up to some maximum value …

Counters

all counters have some common fields
- a current value
- a current direction

all counters have some common methods that share a common implementation
- a method to get the value
- a method to get the direction
- **`equals`**, **`hashCode`**, **`toString`**

different counters require different implementations of common methods
- a method to advance the counter in the current direction

Counters

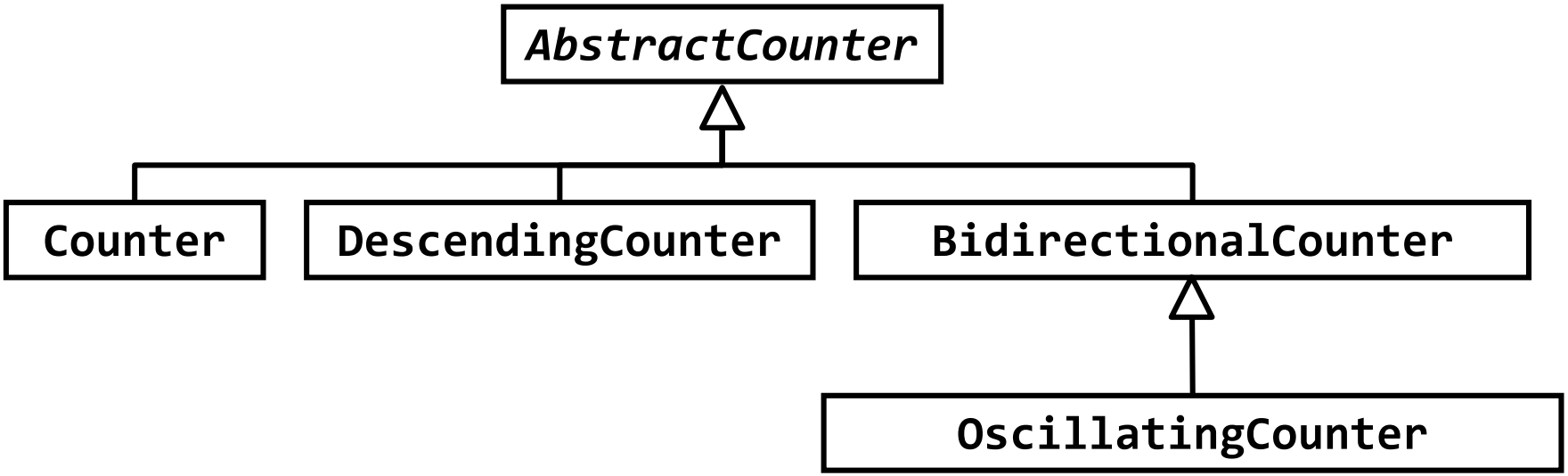
An **inheritance hierarchy** for counters

abstract superclass

- the common fields
- the implementation of common methods
- the declaration of methods that subclasses must provide
  - these are the abstract methods

subclasses

- new methods specific to the subclass
- the implementation of abstract methods specified by the superclass


AbstractCounter
Counter
DescendingCounter
BidirectionalCounter
OscillatingCounter

Interfaces (in Java)

Similar to abstract classes

Contains only methods with empty bodies

Interface methods are by default abstract and public

Interface attributes are by default public, static and final

```
public interface List<E> extends Collection<E>{
        void add(int index, E o);
        boolean contains(Object o);
        E get(int index);
        …
}
```

Interfaces vs. Abstract Classes

Similar to abstract classes/methods
- Cannot be used to create objects
- Interface methods do not have a body

Different from abstract classes
- Classes in Java can **implement** multiple interfaces (but can extend only one class
- An interface cannot contain a constructor (since it cannot be used to create objects)
- On implementation of an interface, you must override all* of its methods (but one can *extend* interfaces)
- *– Interfaces may contain “default” implementations of methods (similar to non-abstract methods in abstract classes; these default methods cannot access object fields (o. state)

https://www.w3schools.com/java/java_interface.asp
https://www.programiz.com/java-programming/interfaces

Static Features

…and inheritance

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
  - can still 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

```
// 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

```
// 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 every time 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 the second time!

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; hides Dog.numCreated

  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

Significant difference between calling a static method and calling a non-static method when dealing with inheritance

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() );
  }
}
```

Dog version

Dog version

Mix version

Komondor version

prints 2

2

1

1

What's Going On?

there is no dynamic dispatch on static methods

The declared type is used, the actual type is ignored

Declared type of **`mutt`** is **`Dog`**

the **`Dog`** version of **`getNumCreated`** that is called

Declared type of **`shaggy`** is **`Dog`**

the **`Dog`** version of **`getNumCreated`** that is called

Hiding Methods

`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

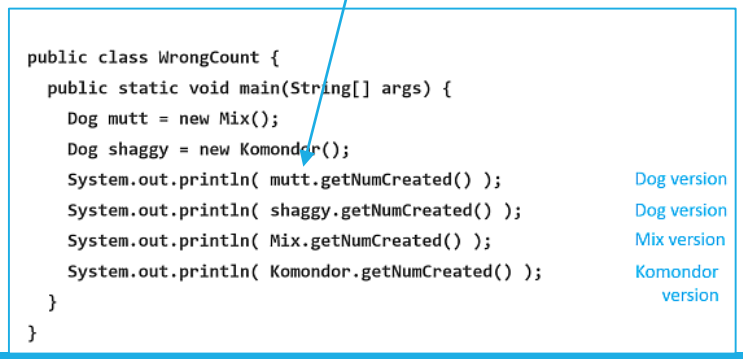
cannot override a static method, can only hide it

hiding static methods is considered bad form because it makes code hard to read and understand

Hiding Methods

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**



Using superclass methods

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)

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

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 original 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();
}
```

size and energy of the dog

breeds of the mix

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;
}
```

use this.energy and this.size to compute the hash code

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;
}
```

use this.energy, this.size, and this.breeds to compute the hash code