Introduction to Interfaces

CS 2110

Announcements

- **Homework 3** out
  - Focus on *refactoring* code
  - Including ideas of
    - *Inheritance*
    - *Abstraction*
    - *Interfaces*
  - We'll cover material on Interfaces today and next class
  - Due: by 11:30pm next Friday.

What is something these things all have in common?

They share a common INTERFACE

Interfaces: a metaphor

Java Allows you to Define Interfaces

Real World (e.g. outlets):

- Using **standardized plugs** and **outlets** allows **reuse** of the power network by any electrical device
- A device implementing a standard North American plug is "*promising*" to support 120V AC (what the power network supplies)

Java (Interfaces):

- Defining an **Interface** allows **reuse** of algorithms and code
- A **class** that implements an interface is "promising" (*contract*) to support methods defined in the interface

Interfaces

The essence of Polymorphism

Polymorphism: Abstract and Concrete classes

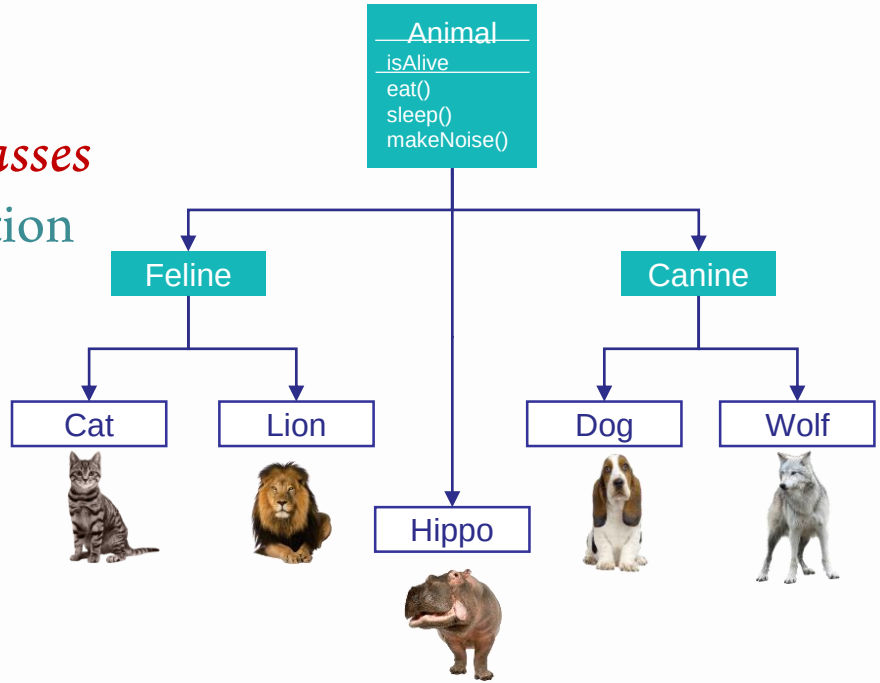
- Shaded boxes are *abstract classes*
  - They need to be *extended*
- Clear (white) boxes show *concrete classes*
  - They need to provide implementation for the *abstract methods*

```
abstract class Animal {
      boolean isAlive;

      abstract public eat();

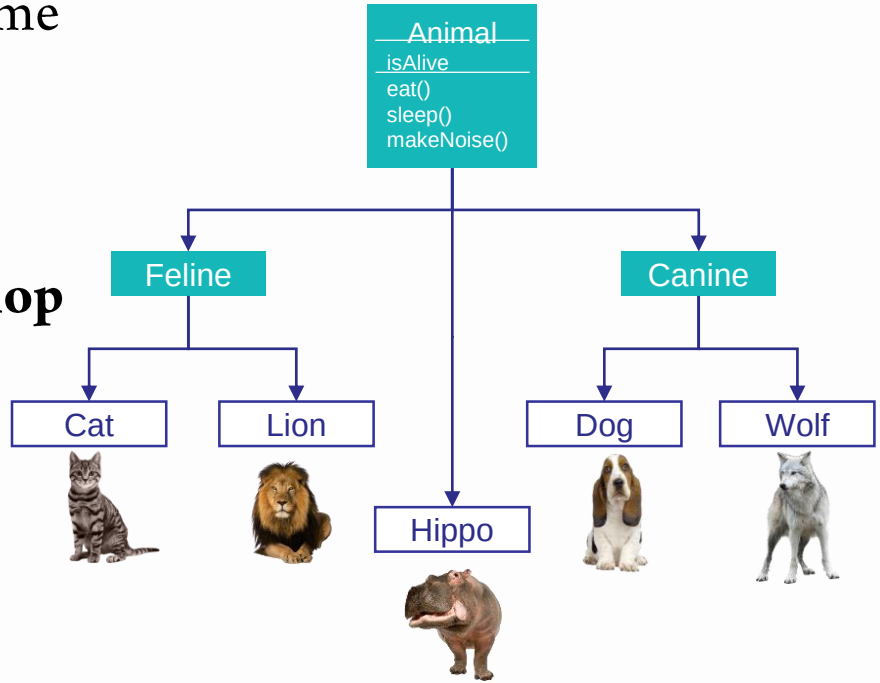
      abstract public sleep();

      abstract public makeNoise();
```



Polymorphism: Changing the Contract

- This class structure works *great* for some applications
  - An animal simulation program
  - A Science Fair tutorial
- What if you want to use it for a **Pet Shop program?**
  - You need Pet *behaviors*
  - E.g. `beFriendly()`, `play()`, …



Polymorphism: How to deal with different behaviors?

1. Implement all methods in the Animal class:

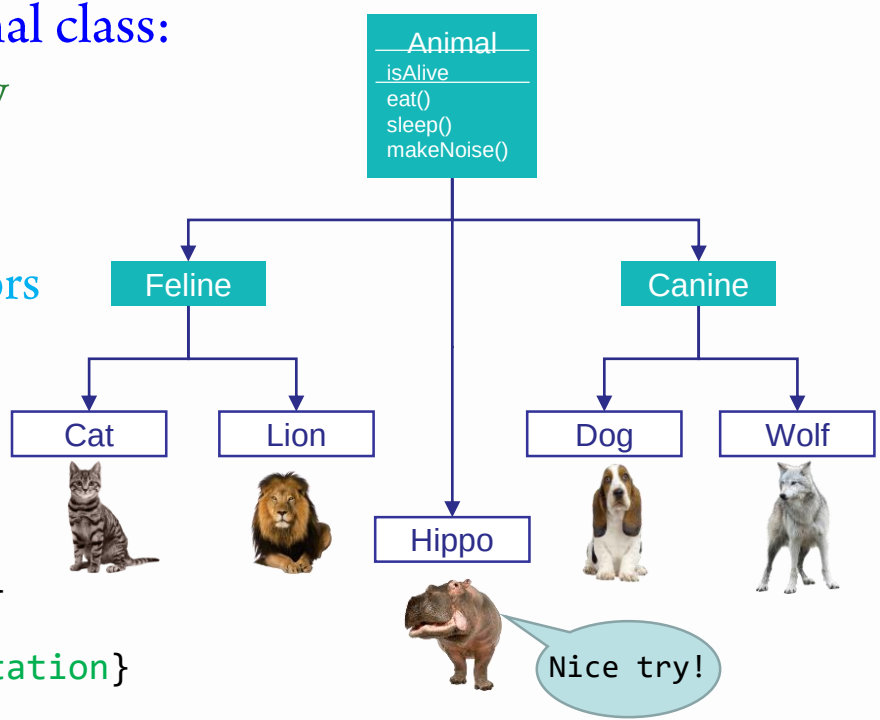
+ All animals inherit behaviors automatically
+ We don't have to edit the subclasses
+ All future animals will have Pet behaviors
– Hippos/Lions/Wolves having Pet behaviors
– Different animals express Pet behaviors

differently so we will need to **edit** them

```
abstract class Animal {
   boolean isAlive;

   public void play() {//implementation}

   public void beFriendly() {//implementation}
}
```



Polymorphism: How to deal with different behaviors?

2 Create abstract methods in the Animal class:

\+ All animals inherit behaviors automatically

\+ Every subclass needs to override the methods

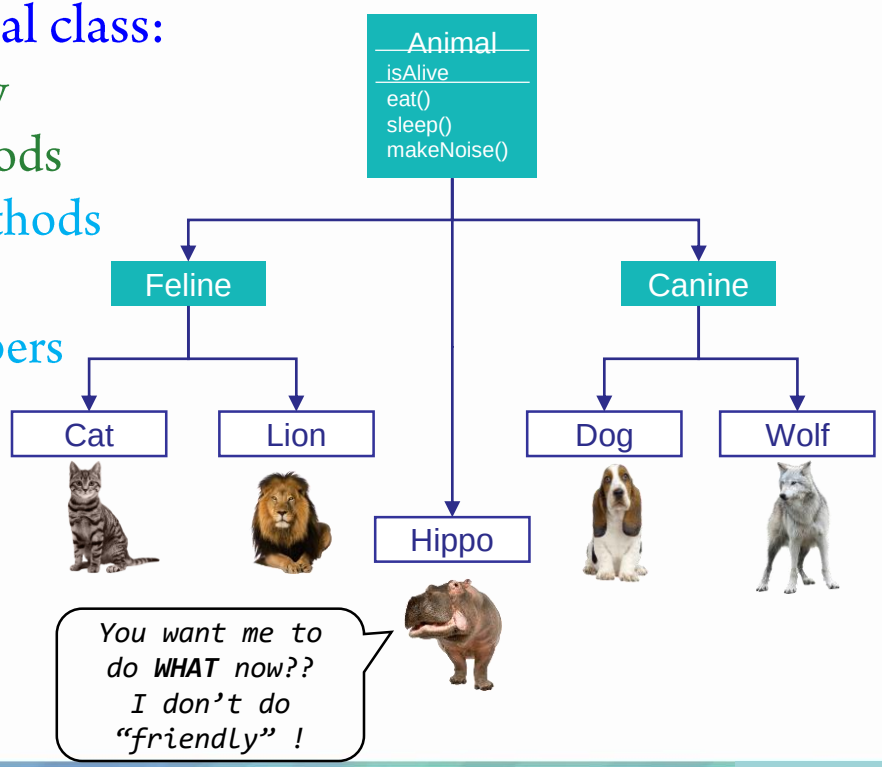
– Even non-pets **need** to implement the methods and have them do NOTHING

– Wrong contract communicated to developers (i.e. Hippos doing Pet things)

```
abstract class Animal {
    boolean isAlive;

    public abstract void play();

    public abstract void beFriendly();
}
```

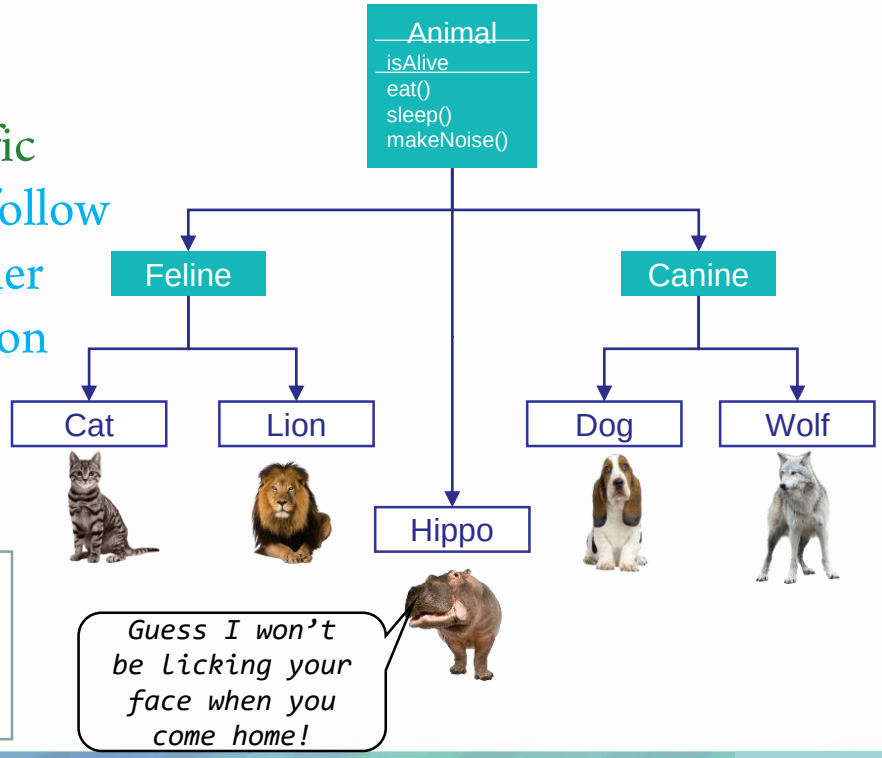


Polymorphism: How to deal with different behaviors?

3 Only pets have Pet methods:

+ Non-pets do not exhibit pet behaviors
+ Behaviors (methods) can be animal-specific
– There is *no contract* for every subclass to follow
– Mistakes can easily happen and the compiler will have no protocol to check implementation

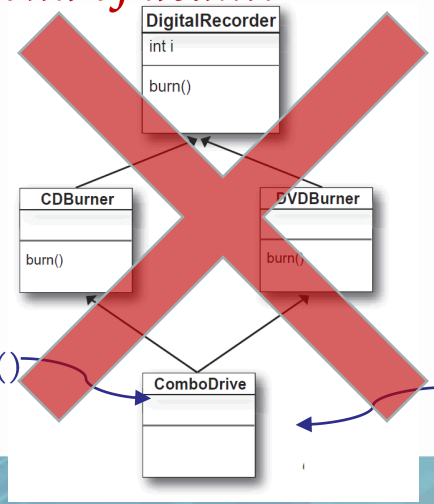
```
abstract class Animal {
   boolean isAlive;
   // Nothing changes in the superclass
}
public class Cat . . . {// A pet class
   public void play() {…}
   public void beFriendly() {…}
}
```



Polymorphism: Multiple Inheritance

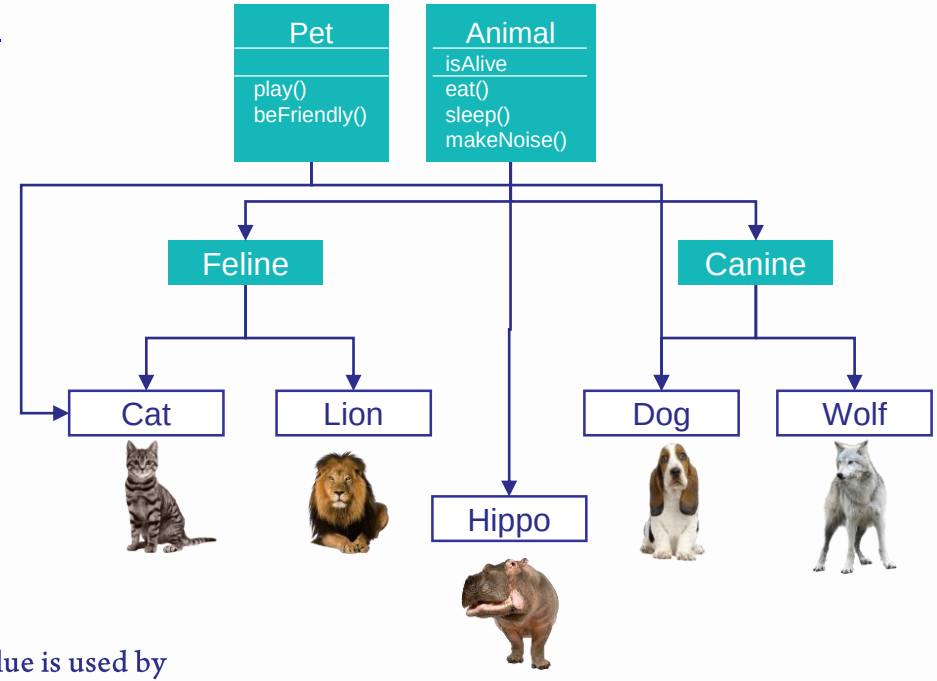
4 Pet animals inherit behavior from two super classes:

+ Pets extend from two super classes
+ Non-pets are not affected
– *Deadly diamond of death!*

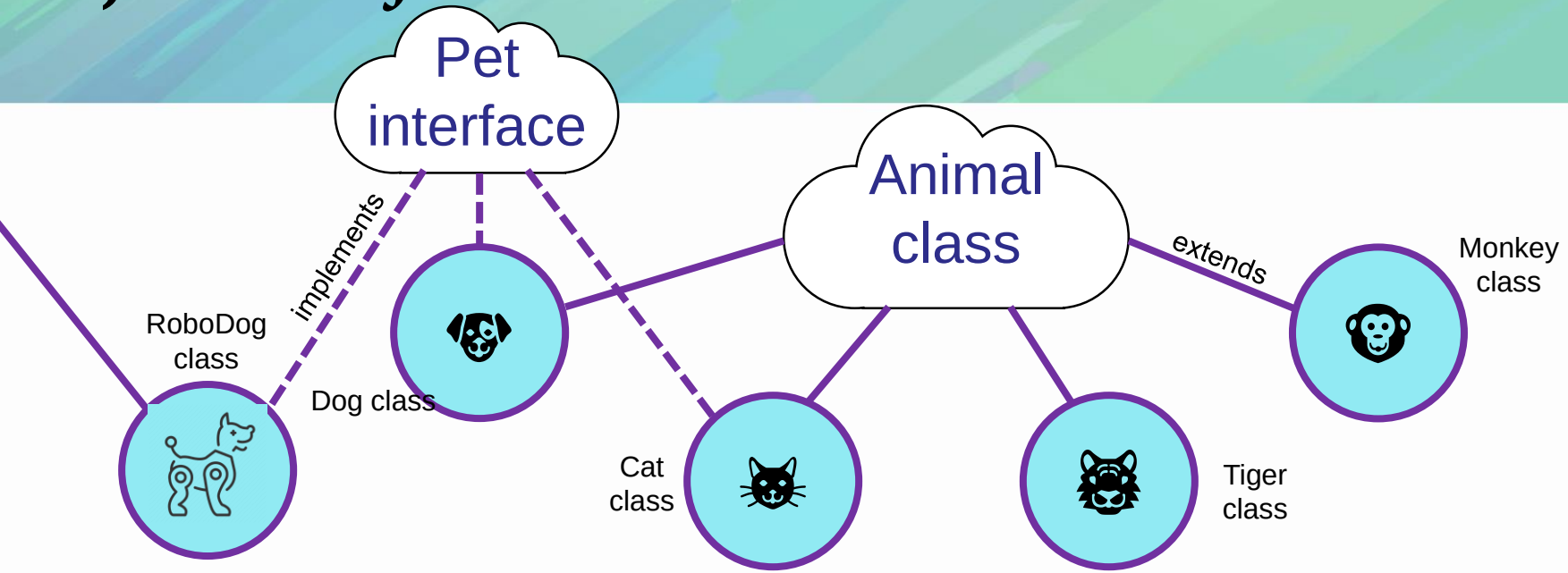


Which burn() method runs when we call burn() on ComboDrive?

Which *i* value is used by ComboDrive if value of *i* is changed by both parents?



Java Interface: 100% abstract class



- *Interfaces* solve the multiple inheritance problem by adding a different *role* to classes
- All methods of an interface are **abstract**, so any class that IS-A Pet must override (write bodies) these methods
- Classes from different inheritance trees can implement the same interface

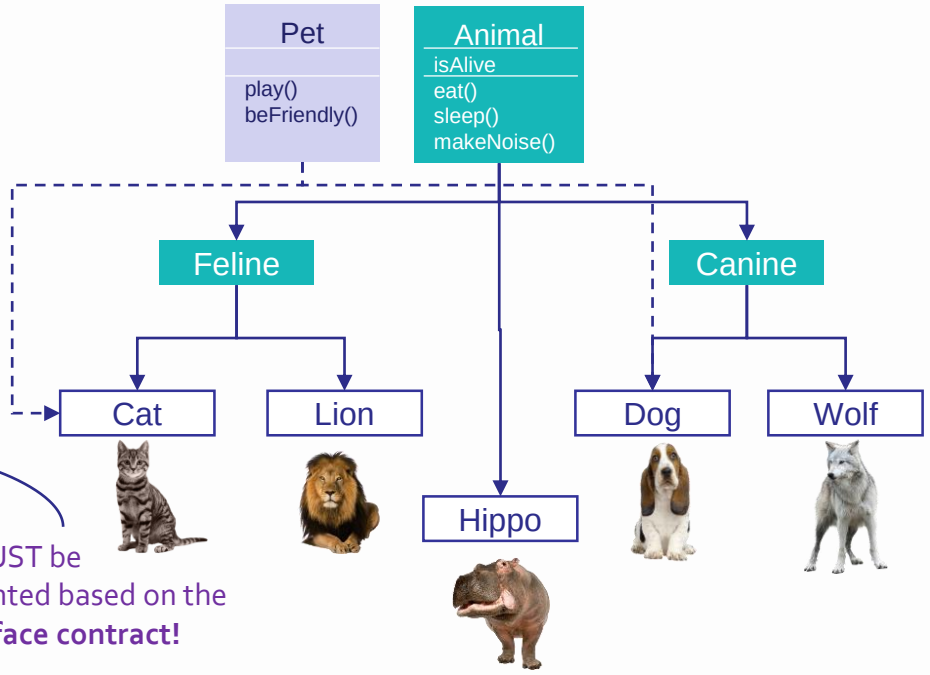
Pet Interface: Adding the Pet role to your pet-animal classes

- Use the key word *implements*

```
public interface Pet {
   public abstract void play();
   public abstract void beFriendly();
}

public class Dog extends Canine
                 implements Pet {
   public void play() {…}
   public void beFriendly() {…}

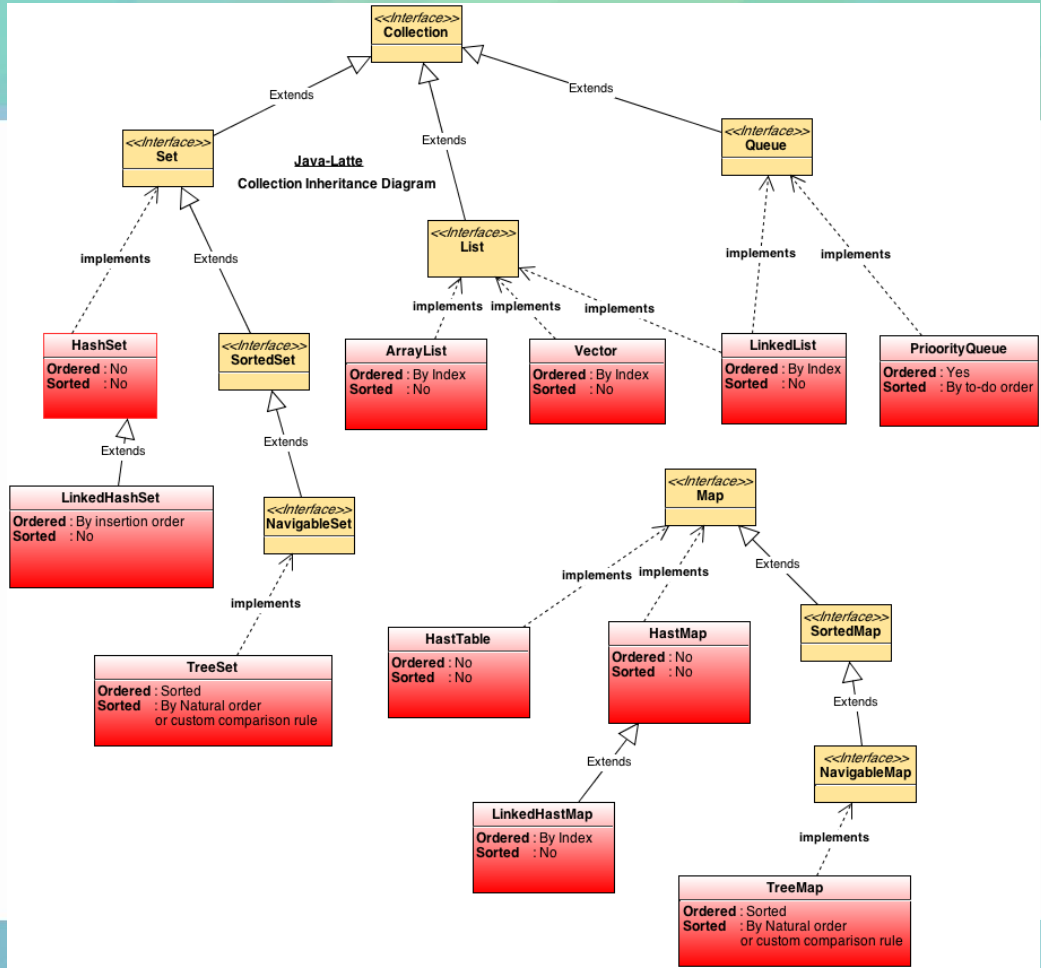
   public void eat() {…}
   public void sleep() {…}
   public void makeNoise() {…}
}
```



Have we seen interfaces before?

- Yes we have!!

- Java Collections Framework
  - List Interface
  - Set Interface
  - Map Interface



Interfaces in Java: Benefits (including increasing maintainability in software development)

- Interfaces support loose-coupling
  - Units of code can be joined using minimum information of other separate components
- Interfaces support high-cohesion
  - Code within a module/unit (class or interface) is related, it belongs together
- Java is designed to support code reuse
  - Multiple classes can make use of an interface (*contract*) regardless of inheritance origin

Important Details about Interfaces

- All methods in an interface are **abstract** (method stubs - no body)
  - They reply on polymorphism

- An interface is a ***reference type*** in Java – it is *similar* to class

- A class **implements** an interface, thereby **inheriting** the abstract methods of the interface (The class **must** implement those abstract methods!)

Important Details about Interfaces

- Single parent, multiple Interfaces!
  - A Java class can ***extend*** only one (parent/super) class
    - This includes an abstract class
    - Which defines **who** you are (IS-A)
      - *Remember the diamond of death?!*
  - A Java class can ***implement*** multiple Interfaces
    - Which defines the **roles** you can play

Important Details about Interfaces

- An **interface** is *different* from a regular class in many ways:
  - All methods are abstract (method stubs) – as mentioned
  - An interface does not contain any **constructors**
  - An interface cannot contain instance fields/attributes
    - Only fields allowed must be declared both "**static**" and "**final**"
    - Although, you can define constants
  - **You cannot instantiate an interface**
    - Not allowed: `MyInterface i = new MyInterface(…);`

Interfaces Form a Contract!

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.

If your class claims to implement an interface, ***all*** methods defined by that interface **must** appear in its source code before the class will successfully compile.

Choosing between classes, subclasses, abstract classes, and interfaces

Type	Description
Class	Make a **CLASS** that does not extend anything (besides Object) when your new class ***does not pass the IS-A test*** for any other type
Subclass	Make a **SUBCLASS** (i.e., extend another class) ONLY when you need to make a ***more specific version*** of a class and need to override or add new behaviors
Abstract class	Use an **ABSTRACT CLASS** when you want to define a ***template (contract)*** for a group of subclasses, and you have at least some implementation code that all subclasses should use (not all abstract methods); also when you want to ensure that ***no objects of that type can be instantiated***
Interface	Use an **INTERFACE** when you want to define a ***role*** that other classes can play (i.e., subclass implements interface), regardless of where these classes are in the inheritance tree

In-Class Activity: Interfaces

Solids

- Work with a partner
- Submit individually on Collab by the end of class or by 11:30pm tonight

In-Class Activity: Solids (Interfaces)

- You are given 4 classes:
  - ***Cuboid***, ***Sphere***, ***Octahedron***, and ***Pyramid***
- You will need to create an interface called ***Solids*** that requires three methods:
  - `getVolume()`, `getColor()`, and `getName()`
- Modify each of the 4 classes (***Cuboid.java***, etc.) to implement the Solids interface in a way that is appropriate for the given shape
  - *Don't worry – we give you hints!*
- After that, run the ***SolidsTest*** class that is provided and see if you get the desired output

Want More?

You're responsible for this material too

This section simply provides additional examples

** Animal Interface example

** Timekeeper Interface example and explanation

Example of an Interface

```
/* File name : Animal.java */
public interface Animal {
   public void eat();
   public void travel();
}
```

Notice the semi-colon at the end of the method headers!

→*Method "stubs"*

Example of Implementing Class

```
/* File name : Mammal.java */
public class Mammal implements Animal {

  public void eat() {  // must implement!
    System.out.println("Mammal eats");
  }

  public void travel() { // must implement!
    System.out.println("Mammal travels");
  }

  public int noOfLegs() { // other method
    return 4;
  }

  public static void main(String args[]) {
    Mammal m = new Mammal();
    m.eat();
    m.travel();
  }
}
```

Any other class that *implements* Animal can also "eat" and "travel"

Output:
Mammal eats
Mammal travels

What's an Interface?

- When defining a class, state that it "**implements**" an interface.
  [Meets the specifications defined in that interface (like a 120v power plug]
- E.g.

```
public class Watch implements TimeKeeper {…}
```

- What does TimeKeeper do?
  - A set of methods that any implementing class must include
  - TimeKeeper interface **does not** define how these methods are coded
  - Watch (the ***implementing*** class) is promising to include those in its definition

TimeKeeper Example

- A TimeKeeper Interface defined with its 2 methods:
  - **getTime()** and **set Time()**
- Now a **Watch** class is declared to implement TimeKeeper
  - Watch is **promising** to have these 2 methods as part of its definition (getTime() and setTime())
  - Watch can implement those how ever it wants
- It means Watch, by implementing TimeKeeper, can handle the TimeKeeper role

Interface gives an Object another Type

- With this definition:
  ```
  public class Watch implements TimeKeeper {
  ```
- You can think of **Watch** in these ways:
  - You can treat a Watch object as a **TimeKeeper**
  - A Watch object can do "TimeKeeper things"
  - A Watch object can be used anywhere a TimeKeeper is legal to use
- A **Watch** object has more than one type
  - It's a **Watch** (defined by its own class **Watch**)
  - It's a **TimeKeeper** (defined by the ***interface* TimeKeeper**)
  - *(It is also the generic type **Object**, too!)*

Interface gives an Object another Type

- Interfaces are legal **Java types**. Therefore they can be used
  - To declare variables
    - `TimeKeeper myWatch = new Watch();`
  - As a parameter type (passing a variable to a method)
    - `public void checkTimeKeeper(TimeKeeper t) {…}`
  - As a return type
    - ```
      public TimeKeeper buildNewThing() {
              …
              return t;
       }
      ```
      ```

```
public class Watch implements TimeKeeper {
```

Method Type

- If you have a **method** that took a **TimeKeeper** as a parameter
  - A **Watch** could be used as a parameter to that method, because a **Watch** plays the role of a **TimeKeeper**

- → You can think of the Watch object as having more than one type. (Every Watch also fills the role of a Time Keeper)

Watch Fulfills Multiple Types

```
public class TestingTypes{
      public static void main(String[] args){
          Watch w = new Watch(…);
           // The Watch class implements TimeKeeper
          System.out.println(w instanceof Watch);
          System.out.println(w instanceof TimeKeeper);
          System.out.println(w instanceof Object);
      }
}
```

Output:
true
true
true