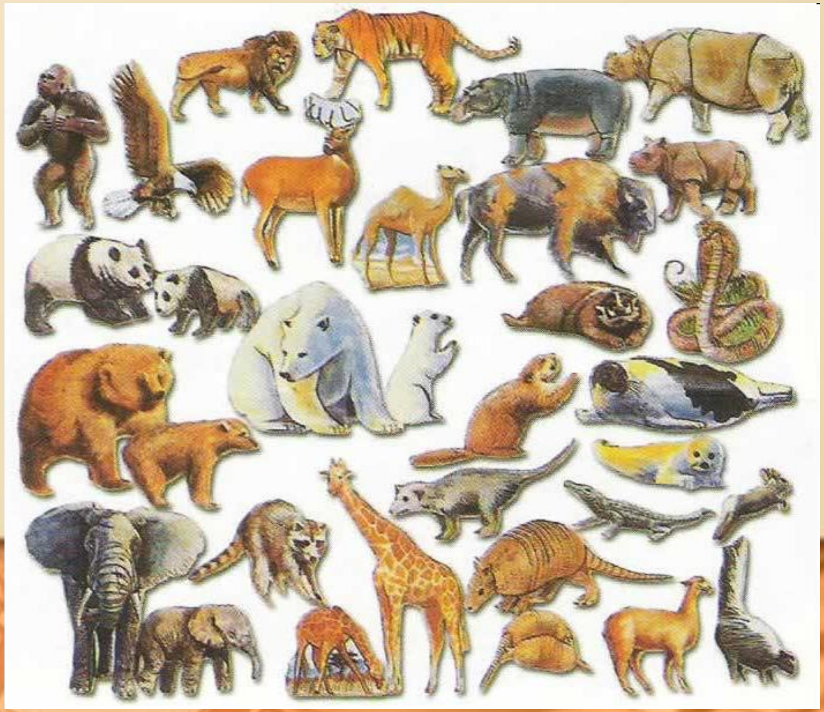
Introduction to Computer Science
Reichman University

Lecture 12-1

Interface Inheritance



Inheritance

Background

- *Inheritance* is a major feature of object-oriented programming
- We'll give an introduction to inheritance, focusing on main concepts and techniques.

Two forms of inheritance

- Class inheritance (last lecture)
- Interface inheritance (this lecture)

Lecture plan

<u>Using interfaces for:</u>

- Type hierarchies
- Polymorphism / heterogeneous collections
- Standardizing software
- Multiple inheritance

Type hierarchy

<u>Required</u> (example)

A simple computer game;

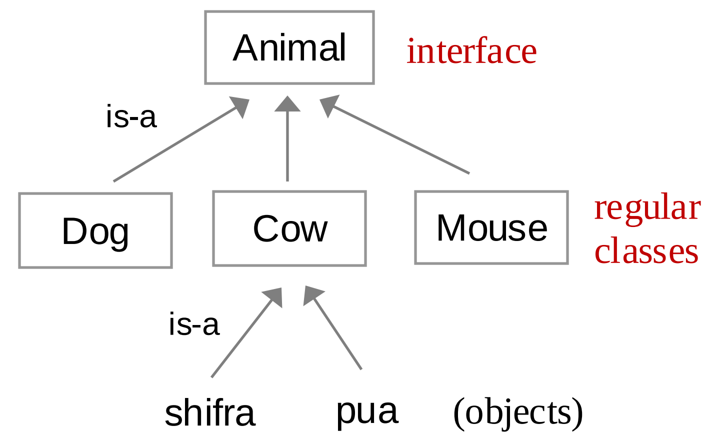
The player sees a random set of animals

The player can:

- Select an animal
- Play the animal’s sound
- Feed the animal
- Do other animal-specific things

Type hierarchy

Type Hierarchy



<u>Required</u> (example)

A simple computer game;

The player sees a random set of animals

The player can:

- Select an animal
- Play the animal's sound
- Feed the animal
- Do other animal-specific things

<u>Software architecture</u>

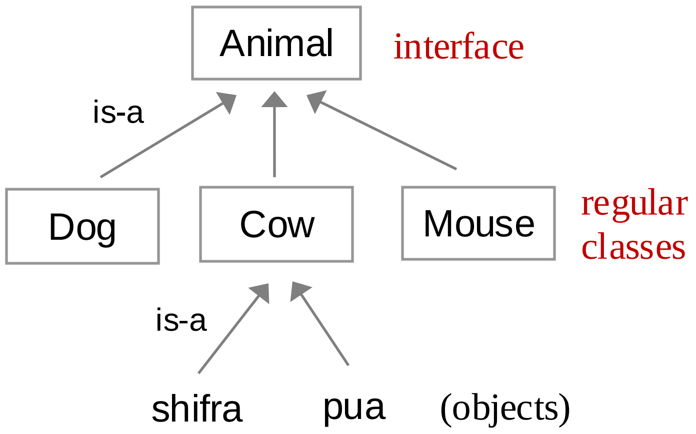
- Declare an `Animal` *type*, representing the common properties and operations (behavior) that each `Animal` object must have
- Declare `Animal` *sub-types*: `Cow`, `Dog`, `Mouse`, ..., each representing the common behavior plus animal-specific behaviors
- Model this *type hierarchy* using *interface inheritance*.

(looks same as class-inheritance, but it's not. Stay tuned)

Type hierarchy: Design

```
/** An Animal (for a simple computer game) */
public interface Animal {

    /** The sound that this animal makes */
    public String sound();

    /** The food that this animal eats */
    public Object eats();
}
```

Type Hierarchy



Software architecture

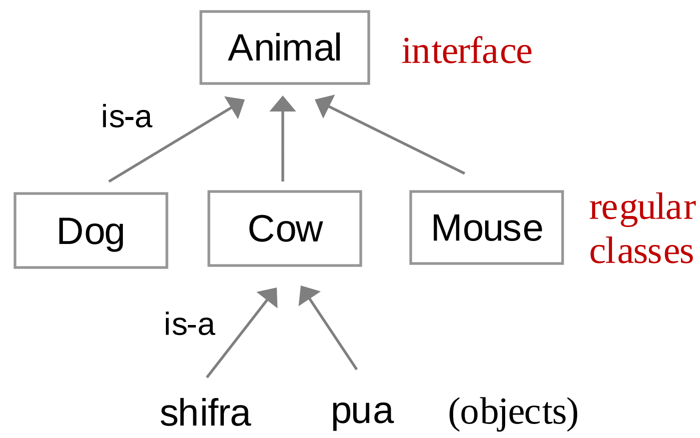
- Declare an `Animal` *type*, representing the common properties and operations (behavior) that each `Animal` object must have
- Declare `Animal` *sub-types*: `Cow`, `Dog`, `Mouse`, …, each representing the common behavior plus animal-specific behaviors
- Model this *type hierarchy* using *interface inheritance*.

Type hierarchy: Design

```
/** An Animal (for a simple computer game) */
public interface Animal {

    /** The sound that this animal makes */
    public String sound();

    /** The food that this animal eats */
    public Object eats();
}
```

```
/** A Cow */

public class Cow implements Animal {

    String food = "hay";
    int weight;

    /** Constructs a cow. */
    public Cow(int weight) {
        this.weight = weight;
    }

    /** The sound of this cow. */
    public String sound() { return "moo"; }

    /** The food of this cow. */
    public String eats() { return food; }

    /** The weight of this cow */
    public double weight() { return weight; }
}
```

Type Hierarchy



Software architecture

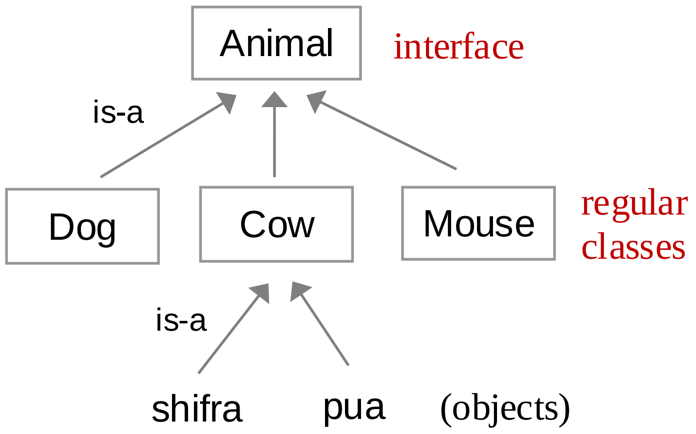
- Declare an `Animal` *type*, representing the common properties and operations (behavior) that each `Animal` object must have
- Declare `Animal` *sub-types*: `Cow`, `Dog`, `Mouse`, ..., each representing the common behavior plus animal-specific behaviors
- Model this *type hierarchy* using *interface inheritance*.

Type hierarchy: Design

```
/** An Animal (for a simple computer game) */
public interface Animal {

    /** The sound that this animal makes */
    public String sound();

    /** The food that this animal eats */
    public Object eats();
}
```

```
/** A Cow */

public class Cow implements Animal {

    String food = "hay";
    int weight;

    /** Constructs a cow. */
    public Cow(int weight) {
        this.weight = weight;
    }

    /** The sound of this cow. */
    public String sound() { return "moo"; }

    /** The food of this cow. */
    public String eats() { return food; }

    /** The weight of this cow */
    public double weight() { return weight; }
}
```

Type Hierarchy



```
// Client code, in any class:

Cow shifra = new Cow();

System.out.println(shifra.sound());
```

```
moo
```

Type hierarchy: Design

```
/** An Animal (for a simple computer game) */
public interface Animal {

    /** The sound that this animal makes */
    public String sound();

    /** The food that this animal eats */
    public Object eats();
}
```

```
/** A cow */

public class Cow implements Animal {

    String food = "hay";
    int weight;

    /** Constructs a cow. */
    public Cow(int weight) {
        this.weight = weight;
    }

    /** The sound of this cow. */
    public String sound() { return "moo"; }

    /** The food of this cow. */
    public String eats() { return food; }

    /** The weight of this cow */
    public double weight() { return weight; }
}
```

<u>An interface:</u>

- Consists only of *abstract methods*
- Has a regular API
- Compiled just like a class
- BUT: *An interface is not a class*:
  no fields, no constructors, no `new Animal`

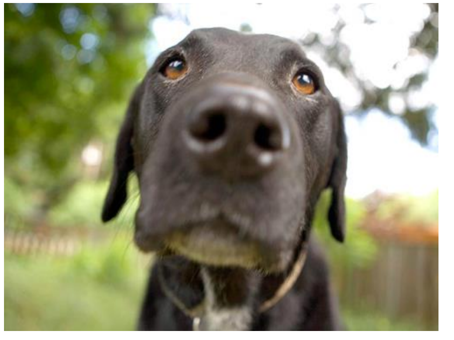
<u>Classes that implement an interface:</u>

- Are regular classes
- Must implement every interface method
  (or get a compilation error)
- Typically define additional methods.

Type hierarchy: Design

```
/** An Animal (for a simple computer game) */
public interface Animal {

    /** The sound that this animal makes */
    public String sound();

    /** The food that this animal eats */
    public Object eats();
}
```

```
/** A dog. */
public class Dog implements Animal {

    String food = "bonzo";

    /** Constructs a dog. */
    public Dog() {
    }

    /** The sound of this dog. */
    public String sound() { return "woof"; }

    /** The food of this dog. */
    public String eats() { return food; }

    /** Causes this dog to fetch the given object */
    public void fetch(Object obj) {
        //code omitted, depends on the game dynamics
    }
}
```



```
// Client code, in any class:

Dog scooby = new Dog();
System.out.println(scooby.sound());
```

```
woof
```

Type hierarchy: Design

```java
/** An Animal (for a simple computer game) */
public interface Animal {
    /** The sound that this animal makes */
    public String sound();
    /** The food that this animal eats */
    public Object eats();
}
```

```java
/** A mouse. */
public class Mouse implements Animal {

    String food = "crumbs";

    /** Constructs a mouse. */
    public Mouse() {}

    /** The sound of this mouse. */
    public String sound() { return "squeak"; }

    /** The food of this mouse. */
    public String eats() { return food; }

    /** Sets the food of this mouse */
    public void setFood(String food) {
        this.food = food; }
}
```

```java
// Client code, in any class:
Mouse mik = new Mouse();
System.out.println(mik.sound());
```

```
squeak
```

Type hierarchy: Design

```
/** An Animal (for a simple computer game) */
public interface Animal {
    /** The sound that this animal makes */
    public String sound();
    /** The food that this animal eats */
    public Object eats();
}
```

```
/** A boa. */
public class Boa implements Animal {
    // The food of this boa
    Animal food;
    /** Constructs a boa and feeds it right away */
    public Boa(Animal a) {
        food = a;
    }
    /** The sound that this boa makes. */
    public String sound() {
        return "((" + food.sound() + "))";
    }
    /** The food of this boa. */
    public Animal eats() { return food; }
}
```

```
// Client code, in any class:
Boa kaa = new Boa(new Mouse());
System.out.println(kaa.sound());
```

```
((squeak))
```

Type hierarchy: Design

```java
/** An Animal (for a simple computer game) */
public interface Animal {
    /** The sound that this animal makes */
    public String sound();
    /** The food that this animal eats */
    public Object eats();
}
```

```java
/** A boa. */
public class Boa implements Animal {
    // The food of this boa
    Animal food;
    /** Constructs a boa and feeds it right away */
    public Boa(Animal a) {
        food = a;
    }
    /** The sound that this boa makes. */
    public String sound() {
        return "((" + food.sound() + "))";
    }
    /** The food of this boa. */
    public Animal eats() { return food; }
}
```

```java
// Client code, in any class:
Boa kaa = new Boa(new Mouse());
System.out.println(kaa.sound());
Boa luna = new Boa(kaa);
System.out.println(luna.sound());
```

```
((squeak))
((((squeak))))
```

Type hierarchy: Design

```java
/** An Animal (for a simple computer game) */
public interface Animal {
    /** The sound that this animal makes */
    public String sound();
    /** The food that this animal eats */
    public Object eats();
}
```

```java
/** A boa. */
public class Boa implements Animal {
    // The food of this boa
    Animal food;
    /** Constructs a boa and feeds it right away */
    public Boa(Animal a) {
        food = a;
    }
    /** The sound that this boa makes. */
    public String sound() {
        return "((" + food.sound() + "))";
    }
    /** The food of this boa. */
    public Animal eats() { return food; }
}
```

Notice the data types

```java
// Client code, in any class:
Boa kaa = new Boa(new Mouse());
System.out.println(kaa.sound());
Boa luna = new Boa(kaa);
System.out.println(luna.sound());
```

```
((squeak))
((((squeak))))
```

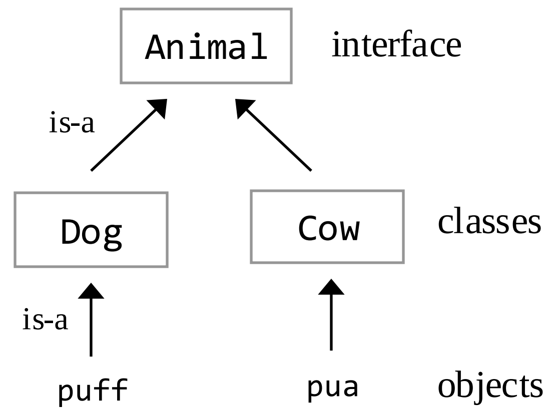
Lecture plan

Using interfaces for:

- Type hierarchies
- Polymorphism / heterogeneous collections
- Standardizing software
- Multiple inheritance

Type hierarchy: Use

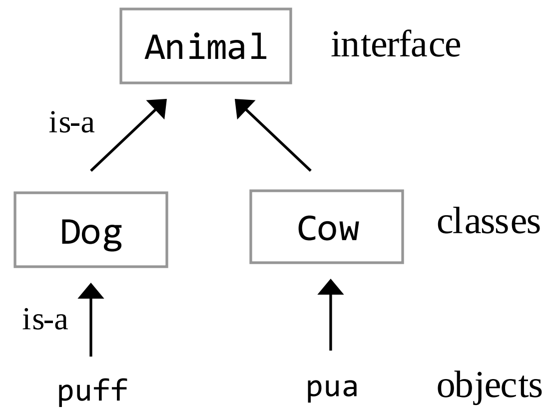
```
// Client code, in some class:
// Creates some animals and plays their sounds.
Cow pua = new Cow(500);
Dog puff = new Dog();
System.out.println(pua.sound());   // moo
System.out.println(puff.sound()); // woof
```



Type hierarchy: Use example 1

```
// Client code, in some class:
// Creates some animals and plays their sounds.

Cow pua = new Cow(500);
Dog puff = new Dog();
System.out.println(pua.sound());   // moo
System.out.println(puff.sound()); // woof

sounds(pua, 2);
sounds(puff, 3);

// Casuse the given animal to make its sound, n times
private static void sounds(Animal a, int n) {
   for (int i = 0, i++, i < b) {
      System.out.println(a.sound());
   }
}
```

```
moo
moo
woof
woof
woof
```



<u>Polymorphic processing</u>

- `sounds` is a *polymorphic method*: The type of one of its arguments is an interface;
- Different objects that implement this interface may do different things.
- Making the argument of this method `Animal` is an example of *upcasting*.

Type hierarchy: Use example 2

```
// Creates a heterogeneous array of Animal objects:
Animal[] animals = { new Cow(500), new Dog(), new Mouse() };

for (Animal a : animals) {
    System.out.println(a.getClass().getName()+ " goes " + a.sound());
}
```

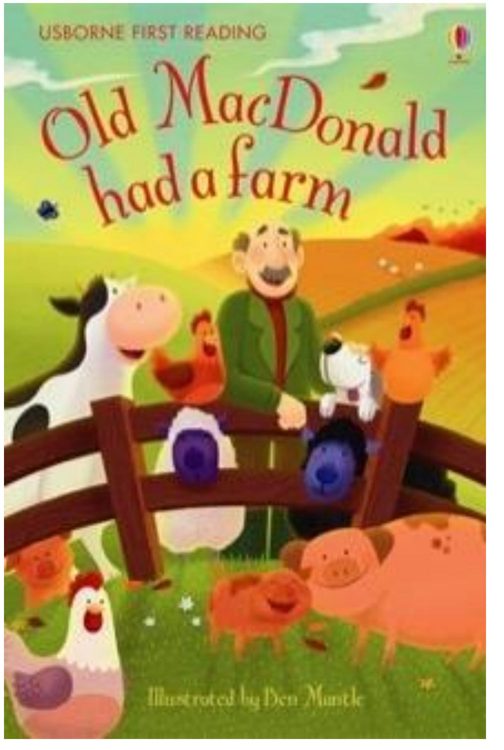
```
Cow goes moo
Dog goes woof
Mouse goes squeak
```

`getClass()`: an `Object` method, returns a `Class` object that represents the class to which this object belongs

`getName()`: a `Class` method, returns the name of this class, as a string

Type hierarchy: Use example 3

```
Old MacDonald had a farm, E-I-E-I-O
And on this farm he had a Cow, E-I-E-I-O
With a moo here and a moo there
Here a moo there a moo
Everywhere a moo moo
Old MacDonald had a farm, E-I-E-I-O

Old MacDonald had a farm, E-I-E-I-O
And on this farm he had a Dog, E-I-E-I-O
With a woof here and a woof there
Here a woof there a woof
Everywhere a woof woof
Old MacDonald had a farm, E-I-E-I-O

Old MacDonald had a farm, E-I-E-I-O
And on this farm he had a Mouse, E-I-E-I-O
With a squeak here and a squeak there
Here a squeak there a squeak
Everywhere a squeak squeak
Old MacDonald had a farm, E-I-E-I-O
```



<u>Task:</u>

- Produce an Old MacDonald song for a farm of animals
- Let the user decide which animals will be in the farm / song.

Type hierarchy: Use example 3

```
public static void main(String[] args) {
    // Put here any animal you want
    Animal[] farm = { new Cow(500), new Dog(), new Mouse() };
    // Produces the song, one verse for each animal in the farm
    for (Animal animal : farm)
        oldMacDonaldVerse(animal);
}
private static void oldMacDonaldVerse(Animal animal) {
    String eieio = ", E-I-E-I-O";
    System.out.println("Old MacDonald had a farm" + eieio);
    System.out.println("And in this farm he had a " + animal.getClass().getName() + eieio);
    String sound = animal.sound();
    System.out.println("With a " + sound + " here and a " + sound + " there" );
    System.out.println( "Here a " + sound + " there a " + sound);
    System.out.println("Everywhere a " + sound + " " + sound);
    System.out.println("Old MacDonald had a farm" + eieio);
}
```

`farm`: a heterogeneous collection

```
Old MacDonald had a farm, E-I-E-I-O
And on this farm he had a Cow, E-I-E-I-O
With a moo here and a moo there
Here a moo there a moo
Everywhere a moo moo
Old MacDonald had a farm, E-I-E-I-O
```

One example out of the three verses (בתים)

Type hierarchy: Use example 4 (yes, interfaces also have very serious uses...)

```java
/** Logic gate interface */
public interface Gate {
    /** Returns the value of this gate */
    public boolean eval(List<Gate> inputs);

    /** Draws this gate */
    public void draw();

    // More abstract Gate methods
}
```

```java
/** And gate */
public AndGate implements Gate {
    // Fields and constructors ...
    public
    public
    // More A
}
```

```java
/** Or gate */
public OrGate implements Gate {
    // Fields and constructors ...
    public
    public
    // More O
}
```

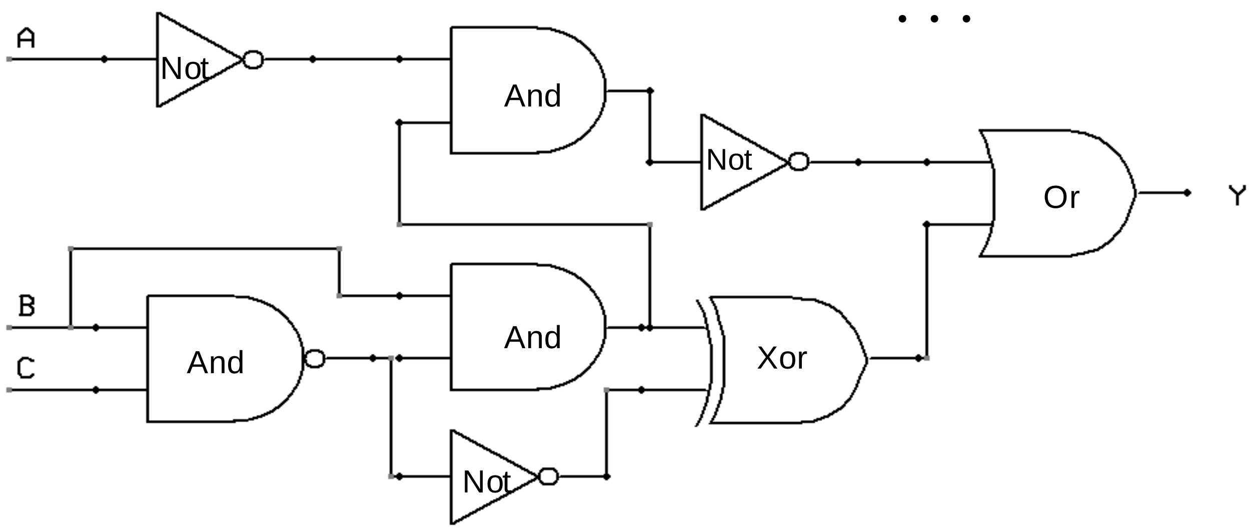
```java
/** Not gate */
public NotGate implements Gate {
    // Fields and constructors ...
    public boolean eval(...) {...}
    public void draw() {...}
    // More NotGate methods
}
```

. . .

Hardware simulator

Used to construct, evaluate, and draw logic circuites (chips):

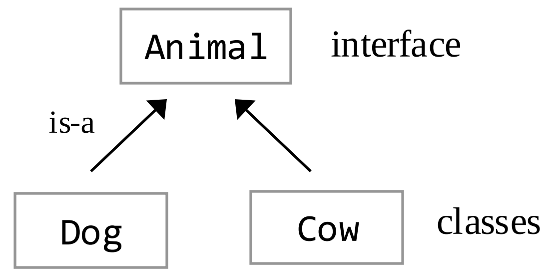
A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



Virtual method calling

```
Animal animal;
Cow pua = new Cow(500);
Dog puff = new Dog();

// The following code serves no purpose, except for illustrating
// a feature of Java known as "virtual method calling"

animal = pua;  // animal points to a Cow object (downcasting)
System.out.println(animal.sound());

animal = puff;  // Now animal points to a Dog object
System.out.println(animal.sound());
```

```
moo
woof
```



Observations

During *compile-time*, we have two identical method calls: `animal.sound()`

During *run-time*, we'll end up calling two different methods: `Cow.sound()` and then `Dog.sound()`

Virtual method calling / AKA Late Binding:

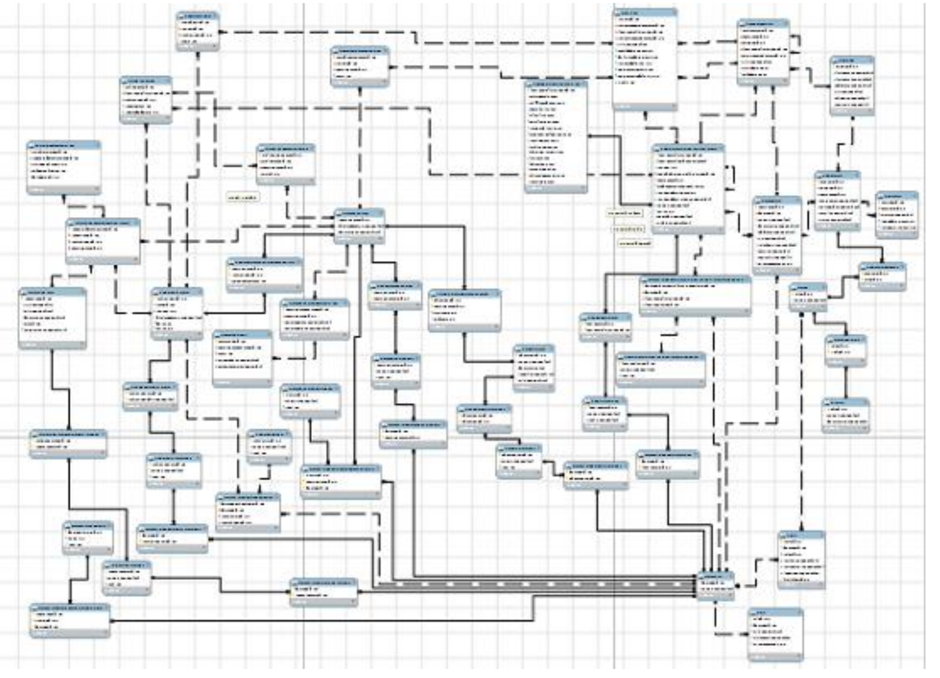
- We see that object types can change (upcasted or downcasted) along the type hierarchy;
- Which method to invoke on an object is determined *during run-time*, according to the object's run-time type.

Lecture plan

<u>Using interfaces for:</u>

- Type hierarchies
- Polymorphism / heterogeneous collections
- Standardizing software
- Multiple inheritance

Using interfaces to standardize systems



The challenge

- Software systems may be based on many classes, built by (possibly) many developers
- The classes are made to interact with each other
- How to reduce complexity?

Interfaces!

- Good systems are characterized by *common* design rules, constraints, and terminology;
- These standards can be managed using *interfaces*.

Using interfaces to standardize systems: Example 1

<u>Example: Object comparison</u>

- How to compare *strings*? *dates*? *fractions*? *sets*? *polynomials*? …
- The comparisons are domain-specific
- However, Java helps standardize object comparisons, using an interface named `Comparable`

Using interfaces to standardize systems: Example 1

From Java's class library:

```
/** Imposes a total ordering on the objects of each class that
    implements it. Objects are compared using the compareTo
    method. Lists and arrays of objects that implement this interface
    can be sorted automatically by Java's Arrays.sort method.  */
public interface Comparable<T> {
   /** Compares this object with the other object.
       Returns a negative integer, zero, or a positive integer as this
       object is less than, equal to, or greater than the other object.  */
  public int compareTo(T other);
}
```

From Java's class library:

```
/** A library of array processing functions */
public class Arrays
  ...
  /** Sorts the given array into ascending order, according to
      the natural ordering of its elements. The array elements
      must implement the Comparable interface. */
  static public void sort(Object[] arr)
  ...
```

My code

```
public class Fraction implements Comparable<Fraction> {
    private int numerator;
    private int denominator;
    ... // Fraction constructors and methods come here
    /** Compares this fraction with the other fraction.
        Returns 0, 1, or -1, as this fraction equals, is greater than,
        or less than, the other fraction. */
    public int compareTo(Fraction other) {
        if (this.equals(other)) return 0;
        int lhs = numerator * other.denominator;
        int rhs = other.numerator * denominator;
        if (lhs > rhs) return 1;
        else return -1;
    }
}
```

Client code

```
// Creates an array of 1000 random fractions
Fraction[] fracs = new Fraction[1000];
for (int i = 0; i < fracs.length; i++)
    fracs[i] = // (code omitted)
// Sorts the fractions
Arrays.sort(fracs);
for (Fraction f : fracs)
    System.out.println(f);
```

```
1/250
23/151
14/84
2/15
5/9
6/7
...
```

If `Fraction` will not implement `comparable`:

We'll get a compilation error (since `Arrays.sort` calls `compareTo`)

Using interfaces to standardize systems: Example 1

From Java's class library:

```
/** Imposes a total ordering on the objects of each class that
    implements it. Objects are compared using the compareTo
    method. Lists and arrays of objects that implement this interface
    can be sorted automatically by Java's Arrays.sort method.  */
public interface Comparable<T> {
   /** Compares this object with the other object.
       Returns a negative integer, zero, or a positive integer as this
       object is less than, equal to, or greater than the other object.  */
  public int compareTo(T other);
}
```

From Java's class library:

```
/** A library of array processing functions */
public class Arrays
   ...
   /** Sorts the given array into ascending order, according to
       the natural ordering of its elements. The array elements
       must implement the Comparable interface. */
   static public void sort(Object[] arr)
   ...
```

Best practice

When designing a class that represents objects that have a natural order (`Fraction`, `Date`, `Point`...), have the class implement Java's `Comparable` interface;

This way, you'll be able to use powerful Java tools that assume that the objects can be compared using a `compareTo` method

Also, the class will look more familiar to programmers who use it.

Client code (another example)

```
// Creates an array of 1000 random dates
Date[] dates = new Date[1000];
for (int i = 0; i < dates.length; i++)
    dates[i] = // (code omitted)

// Sorts the dates
Arrays.sort(dates);
for (Date d : dates)
    System.out.println(d);
```

```
30/07/1746
06/03/1912
01/12/1912
25/01/2019
...
```

(`Date` is a Java class that implements the `Comparable` interface)

Using interfaces to standardize systems: Example 2

Observation: Many classes like `LinkedList`, `ArrayList`, `Queue`, ... represent *collections* of elements.
All these collection types feature element-oriented operations like `add`, `remove`, `compare`, ...

Java's `List<T>` interface features 28 abstract methods for list-oriented operations

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list (optional operation).
void	`add(int index, E element)` Inserts the specified element at the specified position in this list (optional operation).
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	`clear()` Removes all of the elements from this list (optional operation).
boolean	`contains(Object o)` Returns `true` if this list contains the specified element.
boolean	`containsAll(Collection<?> c)` Returns `true` if this list contains all of the elements of the specified collection.
boolean	`equals(Object o)` Compares the specified object with this list for equality.
E	`get(int index)` Returns the element at the specified position in this list.
int	`hashCode()` Returns the hash code value for this list.

etc...

Best practice

When designing a class that represents a collection of elements, consider making the class implement the `List` interface;

Classes that implement commonly-used interfaces are more readable, testable, familiar, and usable.

Lecture plan

Using interfaces for:

- Type hierarchies
- Polymorphism / heterogeneous collections
- Standardizing software
- Multiple inheritance

Multiple inheritance (in a nutshell)

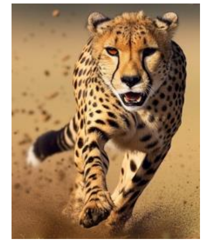
```
public interface Animal {
    /** Animal methods */
}
```

```
public interface Predator extends Animal {
    /** Predator methods */
}
```

```
public interface Amphibious extends Animal {
    /** Amphibious methods */
}
```

```
public class Alligator implements Predator, Amphibious {
    // Alligator fields and constructors declarations
    /** Predator methods */        // mandatory
    /** Amphibious methods */      // mandatory
    /** Alligator methods */       // optional
}
```

An `Alligator` object has the behaviors of both a `Predator` and an `Amphibious`



some animals are predators

some animals are amphibious

some are both predators and amphibious

- A class can implement more than one interface
- Results in a simple form of *multiple inheritance*.

Summary

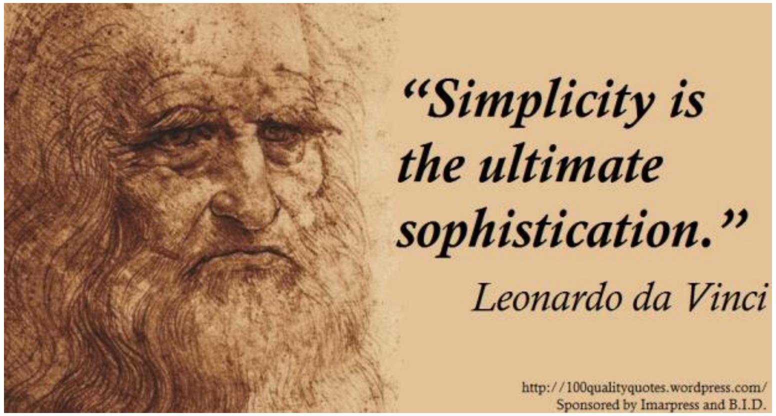
Interfaces provide a relatively simple means for

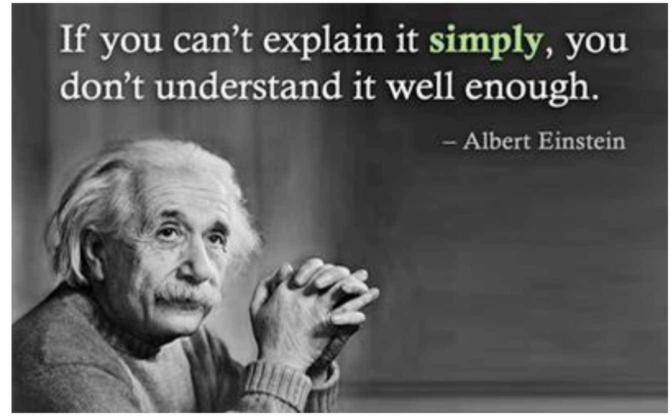
- Modeling type hierarchies
- Polymorphic processing
- Imposing order on software development projects
- Multiple inheritance

And, unlike regular inheritance (sub-classing), interface-based inheritance is ...

Simple

Simplicity





Simple