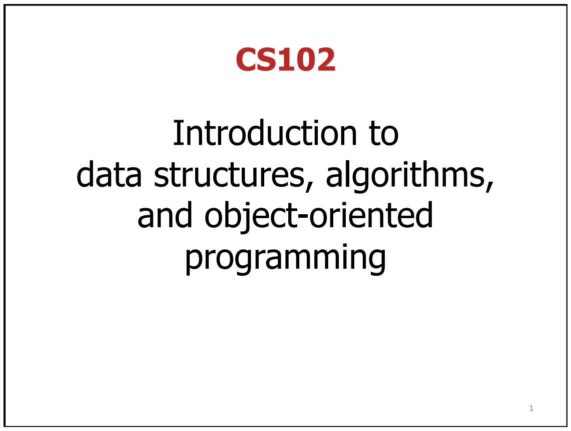
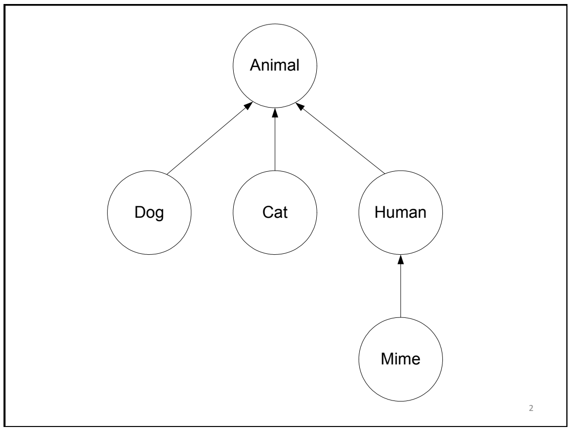
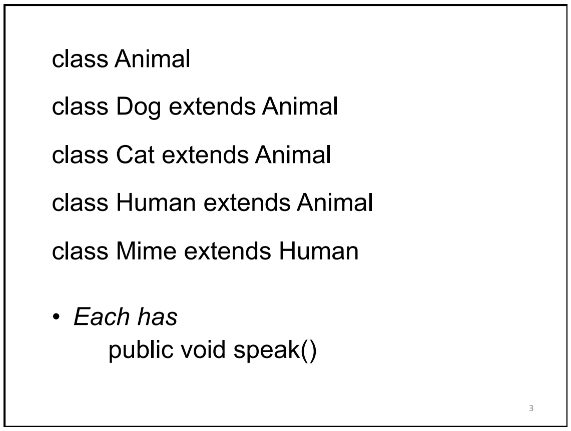


CS102

Introduction to data structures, algorithms, and object-oriented programming





class Animal

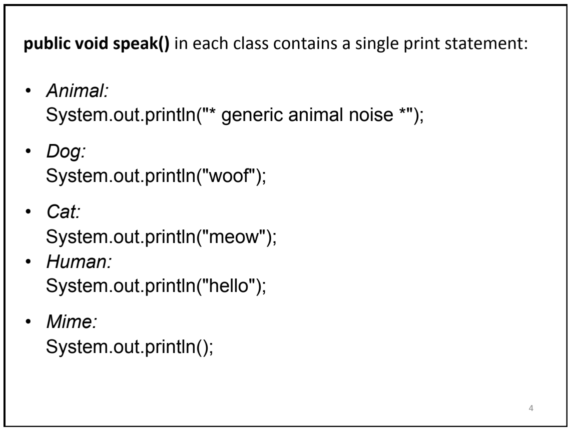
class Dog extends Animal

class Cat extends Animal

class Human extends Animal

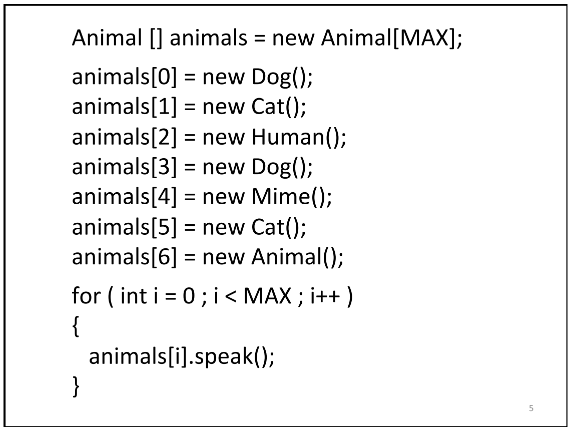
class Mime extends Human

- *Each has*
  public void speak()

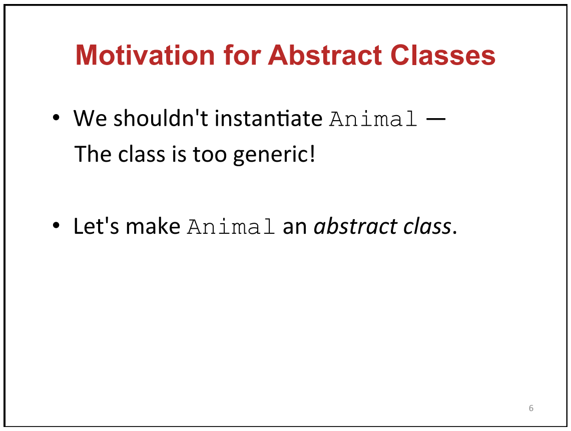


public void speak() in each class contains a single print statement:

- *Animal:*
  System.out.println("* generic animal noise *");
- *Dog:*
  System.out.println("woof");
- *Cat:*
  System.out.println("meow");
- *Human:*
  System.out.println("hello");
- *Mime:*
  System.out.println();



```
Animal [] animals = new Animal[MAX];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Human();
animals[3] = new Dog();
animals[4] = new Mime();
animals[5] = new Cat();
animals[6] = new Animal();
for ( int i = 0 ; i < MAX ; i++ )
{
  animals[i].speak();
}
```



Motivation for Abstract Classes

- We shouldn't instantiate `Animal` —
  The class is too generic!
- Let's make `Animal` an *abstract class*.

```
abstract class Animal
{
//------------------------------------------------------
  abstract public void speak();
//------------------------------------------------------
}
```

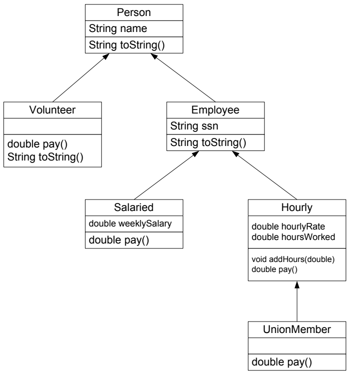
```
Animal [] animals = new Animal[MAX];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Human();
animals[3] = new Dog();
animals[4] = new Mime();
animals[5] = new Cat();
animals[6] = new Cat();
```

```
public void processAnimals(Animal[] ani) {

   for (int i = 0; i < ani.length; i++) {
      ani[i].speak;
   }
}
```

```
static BankAccount processAccount(BankAccount acct)
{
   if (acct instanceof SavingsAccount)
      ((SavingsAccount)acct).addPeriodicInterest();
   else if (acct instanceof CheckingAccount)
      ((CheckingAccount)acct).subtractPenalty();

   System.out.println(acct);
   acct.withdraw(5);
   System.out.println(acct);
   return acct;
}
```



Person
String name
String toString()

Volunteer
double pay()
String toString()

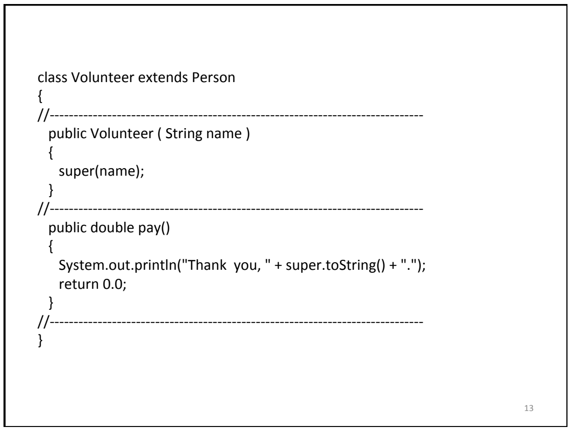
Employee
String ssn
String toString()

Salaried
double weeklySalary
double pay()

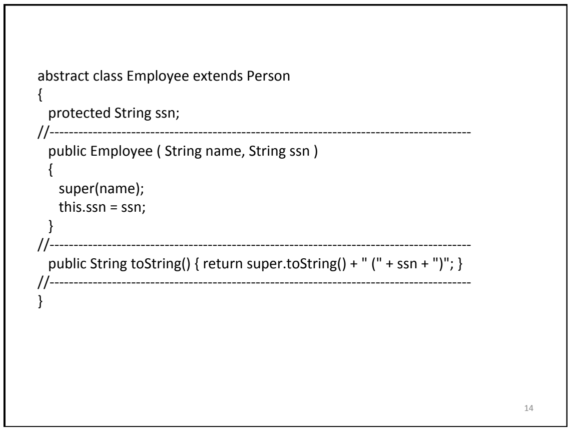
Hourly
double hourlyRate
double hoursWorked
void addHours(double)
double pay()

UnionMember
double pay()

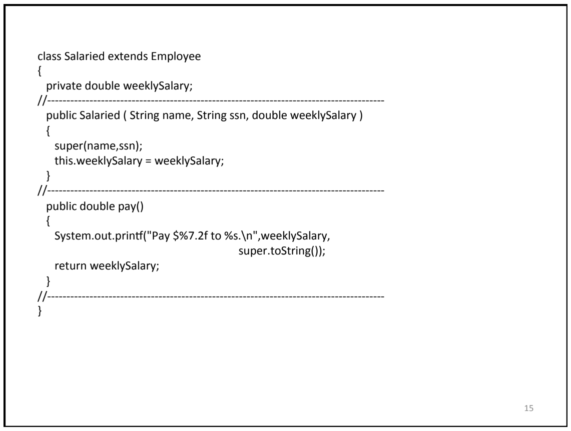
```
abstract class Person
{
  private String name;
//---------------------------------------------------------------------
  public Person ( String name ) { this.name = name; }
//---------------------------------------------------------------------
  public String toString() { return name; }
//---------------------------------------------------------------------
  abstract public double pay();
//---------------------------------------------------------------------
}
```



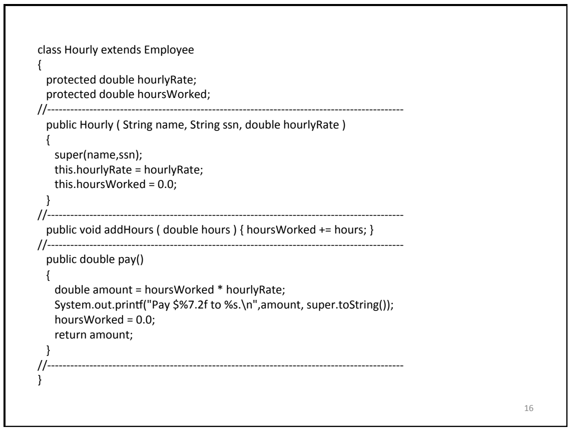
```
class Volunteer extends Person
{
//-------------------------------------------------------------------------------
  public Volunteer ( String name )
  {
    super(name);
  }
//-------------------------------------------------------------------------------
  public double pay()
  {
    System.out.println("Thank  you, " + super.toString() + ".");
    return 0.0;
  }
//-------------------------------------------------------------------------------
}
```



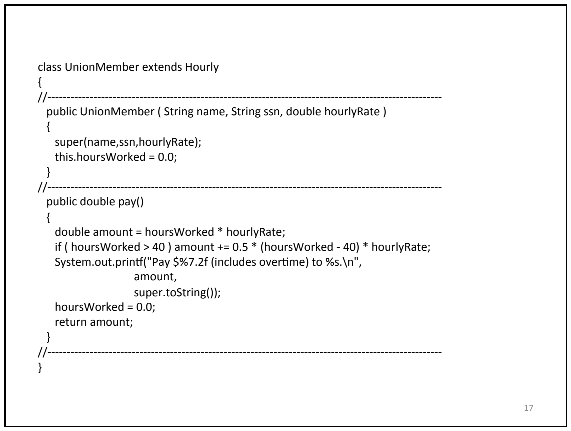
```
abstract class Employee extends Person
{
  protected String ssn;
//------------------------------------------------------------------------------------------
  public Employee ( String name, String ssn )
  {
    super(name);
    this.ssn = ssn;
  }
//------------------------------------------------------------------------------------------
  public String toString() { return super.toString() + " (" + ssn + ")"; }
//------------------------------------------------------------------------------------------
}
```



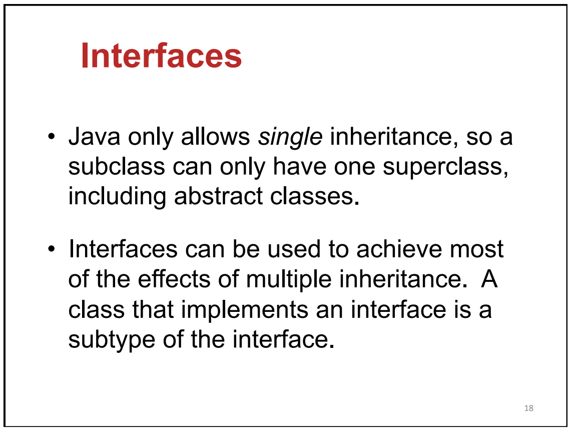
```
class Salaried extends Employee
{
  private double weeklySalary;
//----------------------------------------------------------------------------------
  public Salaried ( String name, String ssn, double weeklySalary )
  {
    super(name,ssn);
    this.weeklySalary = weeklySalary;
  }
//----------------------------------------------------------------------------------
  public double pay()
  {
    System.out.printf("Pay $%7.2f to %s.\n",weeklySalary,
                                              super.toString());
    return weeklySalary;
  }
//----------------------------------------------------------------------------------
}
```



```
class Hourly extends Employee
{
  protected double hourlyRate;
  protected double hoursWorked;
//------------------------------------------------------------------------------------------
  public Hourly ( String name, String ssn, double hourlyRate )
  {
    super(name,ssn);
    this.hourlyRate = hourlyRate;
    this.hoursWorked = 0.0;
  }
//------------------------------------------------------------------------------------------
  public void addHours ( double hours ) { hoursWorked += hours; }
//------------------------------------------------------------------------------------------
  public double pay()
  {
    double amount = hoursWorked * hourlyRate;
    System.out.printf("Pay $%7.2f to %s.\n",amount, super.toString());
    hoursWorked = 0.0;
    return amount;
  }
//------------------------------------------------------------------------------------------
}
```

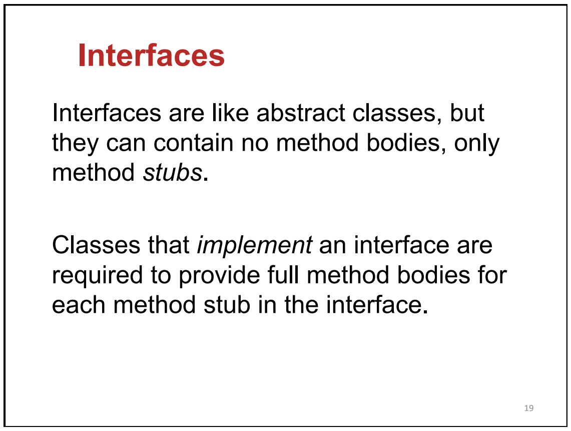


```
class UnionMember extends Hourly
{
//----------------------------------------------------------------------------------------------------
  public UnionMember ( String name, String ssn, double hourlyRate )
  {
    super(name,ssn,hourlyRate);
    this.hoursWorked = 0.0;
  }
//----------------------------------------------------------------------------------------------------
  public double pay()
  {
    double amount = hoursWorked * hourlyRate;
    if ( hoursWorked > 40 ) amount += 0.5 * (hoursWorked - 40) * hourlyRate;
    System.out.printf("Pay $%7.2f (includes overtime) to %s.\n",
                     amount,
                     super.toString());
    hoursWorked = 0.0;
    return amount;
  }
//----------------------------------------------------------------------------------------------------
}
```



Interfaces

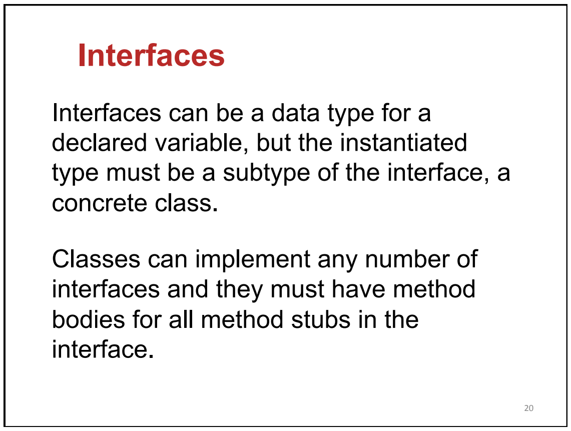
- Java only allows *single* inheritance, so a subclass can only have one superclass, including abstract classes.
- Interfaces can be used to achieve most of the effects of multiple inheritance. A class that implements an interface is a subtype of the interface.



Interfaces

Interfaces are like abstract classes, but they can contain no method bodies, only method *stubs*.

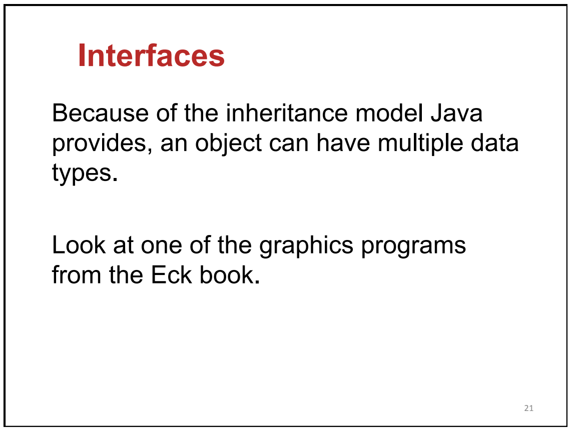
Classes that *implement* an interface are required to provide full method bodies for each method stub in the interface.



Interfaces

Interfaces can be a data type for a declared variable, but the instantiated type must be a subtype of the interface, a concrete class.

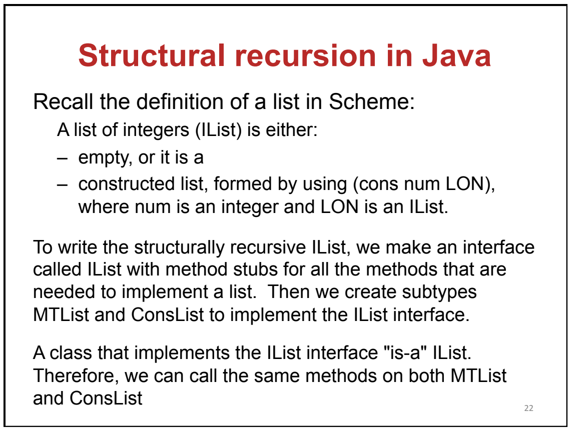
Classes can implement any number of interfaces and they must have method bodies for all method stubs in the interface.



Interfaces

Because of the inheritance model Java provides, an object can have multiple data types.

Look at one of the graphics programs from the Eck book.



Structural recursion in Java

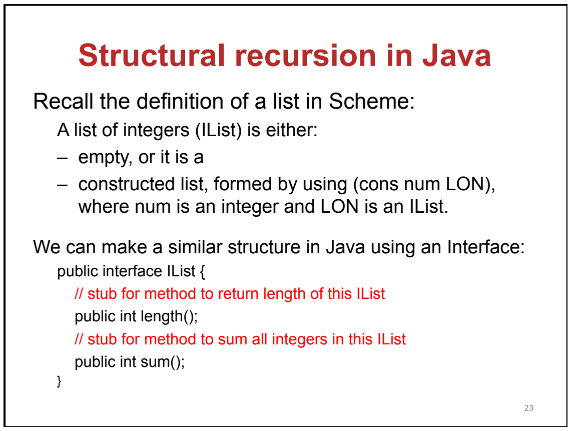
Recall the definition of a list in Scheme:

A list of integers (IList) is either:

- empty, or it is a
- constructed list, formed by using (cons num LON), where num is an integer and LON is an IList.

To write the structurally recursive IList, we make an interface called IList with method stubs for all the methods that are needed to implement a list. Then we create subtypes MTList and ConsList to implement the IList interface.

A class that implements the IList interface "is-a" IList. Therefore, we can call the same methods on both MTList and ConsList



Structural recursion in Java

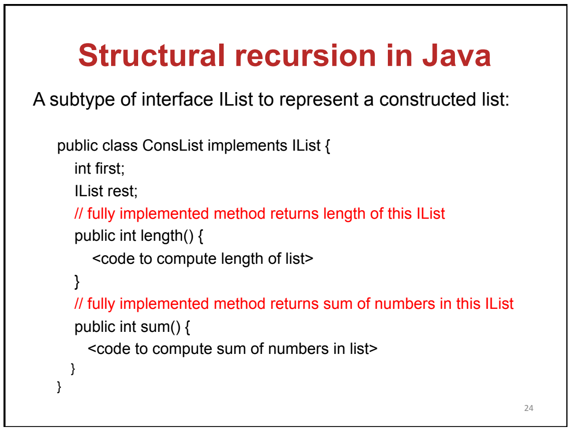
Recall the definition of a list in Scheme:

A list of integers (IList) is either:

- empty, or it is a
- constructed list, formed by using (cons num LON), where num is an integer and LON is an IList.

We can make a similar structure in Java using an Interface:

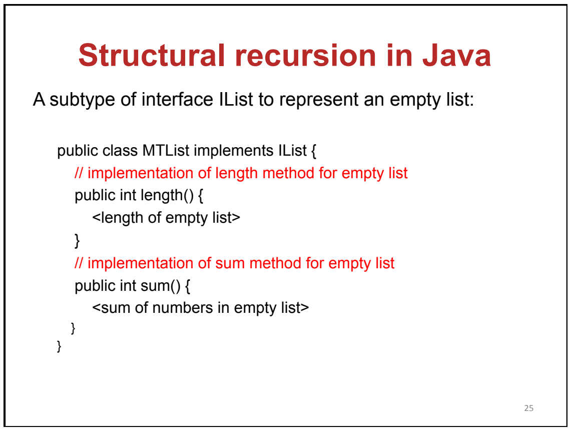
```
public interface IList {
   // stub for method to return length of this IList
   public int length();
   // stub for method to sum all integers in this IList
   public int sum();
}
```



Structural recursion in Java

A subtype of interface IList to represent a constructed list:

```
public class ConsList implements IList {
   int first;
   IList rest;
   // fully implemented method returns length of this IList
   public int length() {
      <code to compute length of list>
   }
   // fully implemented method returns sum of numbers in this IList
   public int sum() {
     <code to compute sum of numbers in list>
   }
}
```



Structural recursion in Java

A subtype of interface IList to represent an empty list:

```
public class MTList implements IList {
   // implementation of length method for empty list
   public int length() {
      <length of empty list>
   }
   // implementation of sum method for empty list
   public int sum() {
      <sum of numbers in empty list>
   }
}
```