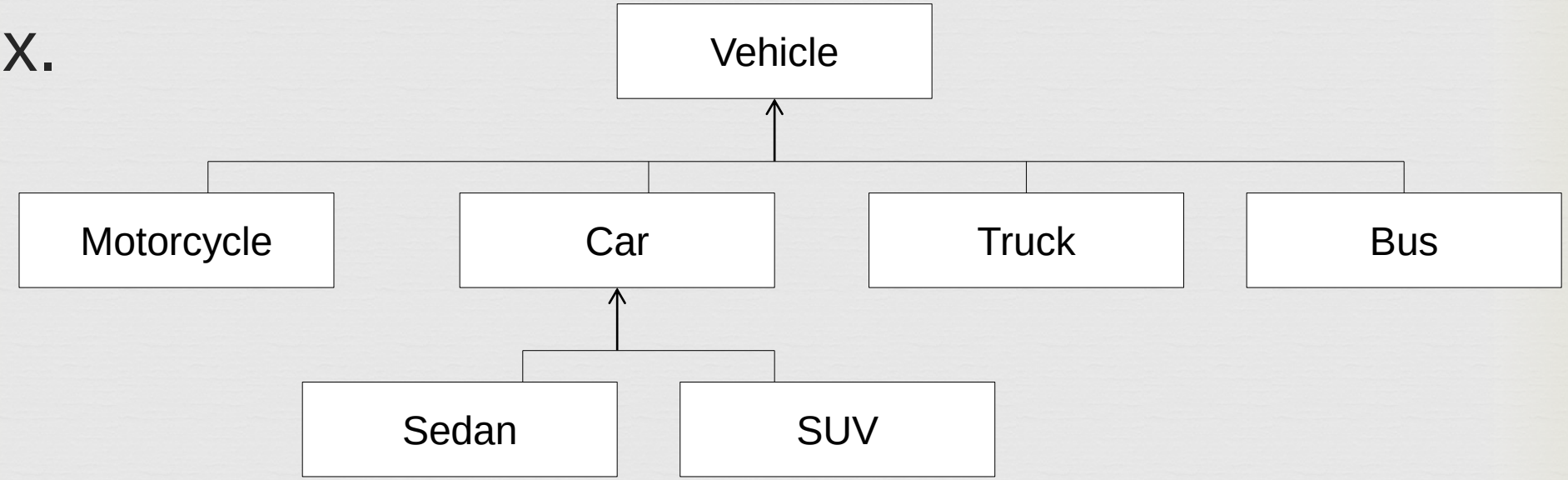
CS 302: Introduction to Programming in Java

Lectures 19&&20

Inheritance Basics

- Inheritance is a relationship between a general object (superclass) and a sub-species of that object (subclass)
- Another key idea of OOP
- Ex.



Reasons for Inheritance

- Invented in 1967 – mimics real-world relationships between objects
- Code reuse
- Substitution principle – you can always use a subclass object when a superclass object is expected

With method: void drive(Vehicle v)

Car myCar = new Car(...);

Motorcycle myMotorcycle = new Motorcycle(...);

drive(myCar); //Ok – cars are a subclass of vehicle

drive(myMotorcycle); //Ok – motorcycles are subclass of vehicle

Implementing Subclasses

- Subclasses inherit all public methods from the superclass
- Can declare new methods unique to the subclass (ex. doWheelie() could be in Motorcycle but not in vehicle)
- Override any inherited methods if their code isn't appropriate for the subclass

Implementing Subclasses Example

```
public class Vehicle
{
  private String licensePlate, make, model, color;
  public Vehicle(....) //initialize instance data
  public void drive()
  public String toString() {
    return liscensePlate + make + model + color;
  }
}
```

Implementing Subclasses Example

```
public class Motorcycle extends Vehicle {
    boolean hasSideCar; //Special motorcycle instance data
    public Motorcycle(....) //initialize instance data
    public void drive() {
       super.drive();
    }
    //override toString() method
    public String toString() {
      return super.toString() + hasSideCar;
    }
    public void doWheelie() //Motorcycle-only method
}
```

Instance Data

- Subclasses have NO access to private instance data of their superclass
- Solution: use super keyword

```
public Motorcycle(String licensePlate, String
    make, String model, String color, boolean
    hasSideCar)
{
    super(licensePlate, make, model, color);
    this.hasSideCar = hasSideCar;
}
```

Instance Data Solution 2

- Use the "protected" keyword instead of "private"
- If a variable is marked "protected" it can be accessed by the class and any of its derived classes (any class which "extends" the class)

In Vehicle:

protected String color; //Now car, motorcycle,

etc. have access to color

Object – The Cosmic Superclass

- All objects automatically descend from the Object class
- We should always override the Object class's equals and toString methods (why?)

Practice

- Use inheritance to implement your own exception
- The exception superclass is called: Exception
- Override the getMessage() method to return a String more unique to your particular exception

Interfaces

- Idea – implement "universal" methods for common problems
- Ex. finding averages, comparing one object to another, etc.

Intro to Interfaces

- Consider 2 methods:

```
public static double average(BankAccount[] objs)
{
  //return avg of all balances in objs
}
public static double average(Country[] objs)
{
  //return avg of the areas of all countries in objs)
}
```

Intro to Interfaces (cont.)

- Note that both methods solve the exact same problem and the code would be very similar
- Only difference is the getter (BankAccount would use objs[i].getBalance(), Country would use objs[i].getArea())
- We can have all classes that need to solve this problem agree on a single method called getMeasure() that returns the instance data needed for computing averages
  - objs[i].getMeasure() // returns a balance if objs[i] was a BankAccount, area if it was a country

Defining an Interface

- Ex.

```
public interface Measurable
{
  double getMeasure();
}
```

- Any object that now has a getMeasure() method "implements" the interface "Measurable"
- Interface methods are always public and have no implementation

Interface Example

```
public class BankAccount implements Measurable
{
  ....//BankAccount stuff
  public double getMeasure()
  {
     return balance;
  }
}
```

Using Interfaces

```
public static double average(Measurable[] objs)
{
   if (objs.length == 0) return 0;
   double sum = 0;
   for (int i = 0; i <  objs.length; i++) {
       sum += objs[i].getMeasure();
   }
   return sum / objs.length;
}
```

Comparable Interface

- Used to compare 2 objects
- Anything that implements Comparable has a compareTo method
- ex. Making BankAccount implement Comparable:

```
public int compareTo(BankAccount other)
{
   return this.balance – other.getBalance();
}
```

CompareTo(Object other)

- Ex. String x = "abc"; String y = "xyz";

if (x.compareTo(y) > 0) { //x is before y}

else if (x.compareTo(y) == 0) { //x = y }

else { //x is after y}

- Always returns an int value
  - 3 possibilities:
    - Return < 0
    - Return 0
    - Return > 0

The Comparable Interface

```
public interface Comparable<T>
{
  public int compareTo(T other);
}
```

- T is the type of object you will compare to

```
public class BankAccount implements
   Comparable<BankAccount> { ...
```

CompareTo

- Useful as many other methods use compareTo
- Ex. Collections.sort() method

```
ArrayList<BankAccount> accounts = new
    ArrayList<BankAccount>();

...

Collections.sort(account); //will sort in ascending
    order
```