

COSC 121: Computer Programming II

Dr. Bowen Hui
University of British Columbia
Okanagan

Quick Review

- **Inheritance** models **IS-A** relationship
- Different from importing classes
- Inherited classes can be organized in a **class hierarchy**
- Special classes to model generic concepts that do not get instantiated are called **abstract classes**

Abstract Class Example

```
public abstract class Animal
{
    protected int numLegs;
    protected int speed;
    public abstract void walks();
    public abstract void eats();
    public int runs()
    {
        // statements to define how fast it runs based on speed
    }
}
```

Abstract Class Example

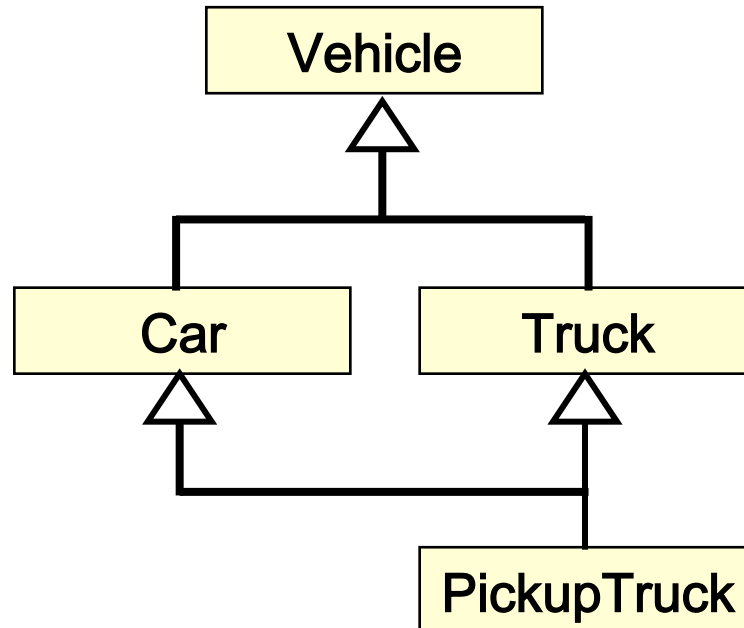
```
public abstract class Animal
{
    protected int numLegs;
    protected int speed;
    public abstract void walks();
    public abstract void eats();
    public int runs()
    {
        // statements to define how fast it runs based on speed
    }
}
```

no constructor

- Abstract methods end with semicolon
- All subclasses must define walks() and eats() or declare them abstract too

Problem of Multiple Inheritance

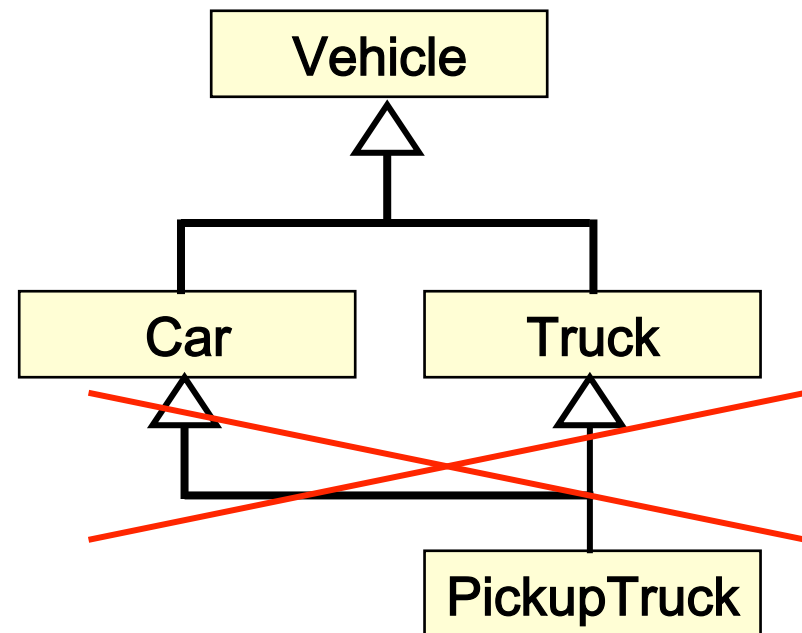
- Recall Java does not allow this:



Problem of Multiple Inheritance

- Recall Java does not allow this:

Car inherits from Vehicle
Truck inherits from Vehicle



PickupTruck **cannot** inherit
from more than one class

Origin of Interfaces

- An OOP programming technique that lets you “conform to” multiple classes
- Avoids the collision problem that arises in multiple inheritance
 - Don’t have any attributes
- A Java **interface** is a group of constants and abstract methods
 - All methods in an interface must be abstract
 - Reserved word `abstract` is not needed

Example

- Example of an interface:

```
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2( double value, char ch );
    public boolean doTheOther( int num );
}
```


Example

- Example of an interface:

```
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2( double value, char ch );
    public boolean doTheOther( int num );
}
```

no constructor

none of the methods
have body definitions

- What does this example remind you of?

Using an Interface

- Suppose you want to define different types of memory storage components
e.g., CD, USB key, etc.
- You have:

```
public interface MemoryInterface
{
    public void writeTo( int location, int value );
    public int readFrom( int location );
    public void loadMemory();
    public int size();
}
```

Implementing an Interface

```
public class SmallDisk implements MemoryInterface
{
    int[] memArray;
    public SmallDisk( int size )
    {
        memArray = new int[size];
    }
    public void writeTo( int location, int value )
    {
        memArray[location] = value;
    }
    public int readFrom( int location )
    {
        return memArray[location];
    }
    public void loadMemory()
    {
        // re-initialize memArray to default values
        memArray = new int[100];
        memArray[0] = 799;
        memArray[1] = 798;
        // etc.
    }
    public int size()
    {
        return memArray.length;
    }
}
```

Implementing an Interface

can have attributes

have constructor

all abstract methods
must be defined

```
public class SmallDisk implements MemoryInterface
{
    int[] memArray;
    public SmallDisk( int size )
    {
        memArray = new int[size];
    }
    public void writeTo( int location, int value )
    {
        memArray[location] = value;
    }
    public int readFrom( int location )
    {
        return memArray[location];
    }
    public void loadMemory()
    {
        // re-initialize memArray to default values
        memArray = new int[100];
        memArray[0] = 799;
        memArray[1] = 798;
        // etc.
    }
    public int size()
    {
        return memArray.length;
    }
}
```

About Interfaces

- An interface *cannot* be instantiated
- All methods are `public abstract`
 - Even if reserved words not used
- All methods must be defined by classes that implement the interface

About Interfaces

- An interface *cannot* be instantiated
- All methods are `public abstract`
 - Even if reserved words not used
- All methods must be defined by classes that implement the interface
- To implement an interface:
 - Use reserved word `implements`
 - Define all the methods from the interface
 - Can define additional attributes and methods

Interfaces vs. Abstract Classes

- Interfaces:
- Abstract classes:

Interfaces vs. Abstract Classes

- Interfaces:
 - No attributes
 - All method are `abstract`
 - All methods are `public`
- Abstract classes:
 - Can have attributes
 - Can have methods with body definitions
 - Methods can have different visibility

When to Use Which?

- Interfaces:
 - Want to guarantee another class will implement a set of methods
 - Choose when: want to specify method signatures to enforce compliance
 - Designed for standardizing communication across classes
- Abstract classes:
 - No guarantee a subclass will override an abstract method
 - Choose when: want subclasses to have a default behaviour
 - Designed for conceptual modeling and to maximize reuse

Predefined Interfaces

- Interfaces other people defined
- `Comparable` interface
 - Contains one abstract method called `compareTo()`
 - Used to compare two objects
 - How to call it:
 - `o1.compareTo(o2)`
 - where `o1` and `o2` are the same types
 - Looks just like any other method call with an object passed in

The Comparable Interface

- Any class can implement Comparable
- Output type: `int`
- Output is ...
 - negative if `obj1` “is less than” `obj2`
 - 0 if the two are equal
 - positive if `obj1` “is greater than” `obj2`
- E.g.:

```
if( obj1.compareTo( obj2 ) < 0 )  
    System.out.println( "obj1 is less than obj2" );
```
- Your class gives meaning to less/greater than

String implements Comparable

- Recall from Ch 5 that `String` class lets us compare strings by lexicographic order
- E.g., “abc” < “bcd”
- You write:

```
String one = "abc";  
String two = "bcd";  
if( one.compareTo( two ) < 0 )  
    System.out.println( one + " is less than "  
        + two );
```

Defining compareTo ()

- Required header definition:

```
public int compareTo( Object o2 )
```

Defining compareTo ()

- Required header definition:
`public int compareTo(Object o2)`
- How come we were able to write:
`one.compareTo(two)`
where `one` and `two` are `Strings`?

Defining compareTo ()

- Required header definition:
`public int compareTo(Object o2)`
- How come we were able to write:
`one.compareTo(two)`
where `one` and `two` are `Strings`?
- Makes use of **polymorphism** (next class)

Defining compareTo ()

- Required header definition:
`public int compareTo(Object o2)`
- How come we were able to write:
`one.compareTo(two)`
where `one` and `two` are `Strings`?
- Makes use of **polymorphism** (next class)
- Inside `compareTo ()`, need to **cast** types
 - Explicitly tells Java what type it really is
 - E.g., for the `String` class, you would have:
`public int compareTo(Object o2)`
`{`
 `String other = (String)o2;`
`}`

Defining compareTo ()



- Required header definition:
`public int compareTo(Object o2)`
- How come we were able to write:
`one.compareTo(two)`
where `one` and `two` are `Strings`?
- Makes use of **polymorphism** (next class)
- Inside `compareTo ()`, need to **cast** types
 - Explicitly tells Java what type it really is
 - E.g., for the `String` class, you would have:

```
public int compareTo( Object o2 )  
{  
    String other = ( String )o2;  
}
```

says “o2 really is a String type”

Shoe Size Comparison

- Compare two shoe sizes based on the following chart:

	Men's Shoe Size	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Women's Shoe Size	5	6	7	8	9	10	11	12	-	-	-	-	-	-

- Note: Men sizes are +2 to women's

Define Shoe class

- Which attributes will you keep track of?

Define Shoe class

- Which attributes will you keep track of?
 - `size`
 - `gender`
- Which methods do you need?

Define Shoe class

- Which attributes will you keep track of?
 - `size`
 - `gender`
- Which methods do you need?
 - `Shoe()`
 - `compareTo()`
- How to access other object's private attributes?

Define Shoe class

- Which attributes will you keep track of?
 - `size`
 - `gender`
- Which methods do you need?
 - `Shoe()`
 - `compareTo()`
- How to access other object's private attributes?
 - `getSize()`
 - `getGender()`

Sample Solution

```
public class Shoe implements Comparable
{
    private int size;
    private int gender;
    private static final int MALE = 0;

    public Shoe( int myGender, int mySize )
    {
        gender = myGender;
        size    = mySize;
    }

    public int getSize()    { return size; }
    public int getGender() { return gender; }
```

Sample Solution (cont.)

```
public int compareTo( Object otherShoe )
{
    Shoe other = ( Shoe )otherShoe;
    int rez = -1;
    if( gender == other.getGender() )
    {
        // within gender, just compare size
        if( size < other.getSize() )
            rez = -1;
        else if( size > other.getSize() )
            rez = 1;
        else
            rez = 0;
    }
    else
    {
        // compare across genders
    }
}
```


Sample Solution (cont.)

```
-  
else  
{  
    // compare across genders  
    // male size always 2 sizes less  
    if( gender == MALE )  
        if( (size+2) == other.getSize() )  
            rez = 0;  
        else if( (size+2) > other.getSize() )  
            rez = 1;  
        else  
            rez = -1;  
    else  
        if( size == ( other.getSize()+2 ))  
            rez = 0;  
        else if( size > ( other.getSize()+2 ))  
            rez = 1;  
        else  
            rez = -1;  
}  
return rez;  
}
```

Testing Your Solution

- Solution has lots of comparisons
- Set up various `Shoe` objects
- Compare them:
 - Write down expected output
(based on your own logic in the comments)
 - Display expected output via `println()`
 - Are they the same?
- Make sure to check all your comparisons

Testing the Sample Solution

```
public class TestShoeSizes
{
    public static void main( String[] args )
    {
        Shoe m3 = new Shoe( 0, 3 );
        Shoe m5 = new Shoe( 0, 5 );
        Shoe m7 = new Shoe( 0, 7 );
        Shoe f5 = new Shoe( 1, 5 );
        Shoe f7 = new Shoe( 1, 7 );
        Shoe f9 = new Shoe( 1, 9 );

        System.out.println( f5.compareTo( f7 ) ); // f5 is smaller
        System.out.println( m5.compareTo( f7 ) );
        System.out.println( m5.compareTo( f5 ) ); // m5 is bigger
        System.out.println( m5.compareTo( f9 ) ); // m5 is smaller
        System.out.println( f7.compareTo( m5 ) );
        System.out.println( f7.compareTo( m3 ) ); // f7 is bigger
        System.out.println( f7.compareTo( m7 ) ); // f7 is smaller
    }
}
```

Interface Hierarchies

- An interface can inherit from another interface
- Example:

```
public interface Sports
{
    public void setHomeTeam( String name );
    public void setVisitingTeam( String name );
}
public interface Volleyball extends Sports
{
    public void setHomeTeamScore( int points );
    public void setVisitingTeamScore( int points );
}
```

- A class that implements `Volleyball` needs to define how many methods?

Interface Hierarchies

- A child interface inherits all abstract methods from the parent
- A class implementing the child interface must therefore define all methods from both interfaces
- Class hierarchies and interface hierarchies don't overlap
 - A class cannot `extends` an interface
 - A class can only `implements` an interface

Multiple Interfaces

- A class can implement multiple interfaces

```
public class ManyThings implements interface1, interface2
{
    // all methods of both interfaces
}
```

- All listed in the same `implements` clause

Simple Example

```
public interface InterfaceA
{
    public abstract void test( int i );
}
public interface InterfaceB
{
    public abstract void test( String s );
}
```

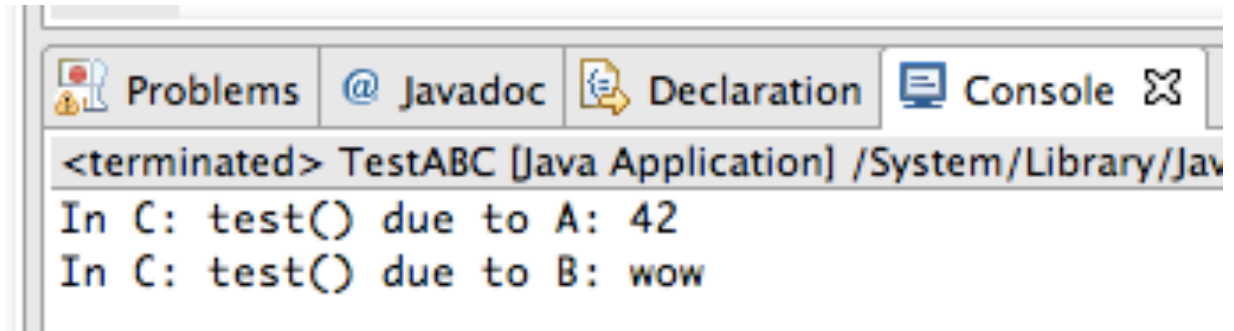
Simple Example

```
public interface InterfaceA
{
    public abstract void test( int i );
}
public interface InterfaceB
{
    public abstract void test( String s );
}
public class ClassC implements InterfaceA, InterfaceB
{
    public void test( int i )
    {
        System.out.println( "In C: test() due to A: " + i );
    }

    public void test( String s )
    {
        System.out.println( "In C: test() due to B: " + s );
    }
}
```


Simple Example

```
public class TestABC
{
    public static void main( String[] args )
    {
        ClassC ccc = new ClassC();
        ccc.test( 42 );
        ccc.test( "wow" );
    }
}
```



Combining `extends` and `implements`

- Recall origin of interfaces:
a way to get around not having multiple inheritance
- A class can extend another class and implement yet other classes
- Example:
`PickupTruck` inherits from `Truck`
`PickupTruck` implements `Car`

PickupTruck Example

```
public class Truck
{
    // define class as usual
}

public interface Car
{
    // constants here (if any)
    // abstract methods here
}

public class PickupTruck extends Truck implements Car
{
    // models Truck as parent class
    // must define all abstract methods required by Car
    // can have additional info: attributes, methods
}
```

Summary of Interface Concepts

- **Interface** makes classes conform to a communication standard
 - All methods are abstract
 - Can include constants definitions
 - Cannot be instantiated
 - No attributes
- New reserved word: `implements`
- Interfaces can extend other interfaces to form an **interface hierarchy**
 - All abstract methods are inherited
- Overcomes not having multiple inheritance
- `Comparable` interface comes with `compareTo()`

Admin

- Next class:
 - Practice between inheritance and interfaces
 - Use them (both) in polymorphism after that
 - Also use arrays and loops – finish A1 Q1 by Thursday
- Labs next week:
 - Practice inheritance