CISC 4700 L01
Network & Client-Server Programming
Spring 2016

Cowell Chapter 8: Inheritance and Polymorphism
Objects: elements of various classes.

Use pre-existing classes where possible.

Otherwise, design your own classes:
   if possible, via inheritance.
May require *polymorphism*:
   redefinition of methods in the original class.
A hierarchy of computer components

Computer components:
  memory
  motherboards
  disks
  monitors

Common characteristics:
  code number
  price

Particular characteristics:
  memory:
    type (e.g., SDRAM), size (e.g., 32M)
  motherboard:
    chipset (e.g., BX), bus speed (e.g., 100 MHz)
  disk:
    size (e.g., 1GB), format (e.g., SATA)
  monitor:
    size (e.g., 17-inch),
    x- and y-resolutions (e.g., 72dpi),
    refresh rate (e.g., 74.5 KHz)
Behaviors associated with all components?
   display details about the component
   give a discount

Note that
   giving a discount is the same for all components
   displaying details is different for different components
   (there are different details)

But any component has code number, price.
   Report these for any kind of component.
Class diagram for computer parts

PCPart
- id: String
- cost: double
+ displayDetails(): void
+ discount(int percentageDiscount): double

Memory
- type: int
- size: int
+ displayDetails(): void

MBoard
- chipset: int
- busSpeed: int
+ displayDetails(): void

Disk
- size: int
- format: int
+ displayDetails(): void

Monitor
- size: int
- xResolution: int
- yResolution: int
- refreshRate: int
+ displayDetails(): void
Implementation of the PCPart class:

```java
public abstract class PCPart {
    String id;             // product identifier
    double cost;           // cost of product

    public void displayDetails() {
        System.out.print("Id number is " + id);
        System.out.println(" Cost is: " + cost);
    }

    public double discount(int percentageDiscount) {
        double theDiscount;
        theDiscount = cost * percentageDiscount / 100.0;
        return theDiscount;
    }
}
```
Note: PCPart is an abstract class. You can't instantiate objects of the PCPart class. Thus

```
PCPart pcpart = new PCPart();
```
is a syntax error.

PCPart: placeholder for common characteristics and behaviors of all the classes.

Notation for abstract class ...

Java: indicated explicitly
C++: indicated via pure virtual function.
The Memory Class

public class Memory extends PCPart {
    int type;       // type of memory
                   //(SDRAM, EDO, FASTPAGE)
    int size;       // memory size in MB

    // kinds of memory, and their descriptions
    static final int SDRAM = 0, EDO = 1, FASTPAGE = 2;
    static String description[] =
        {"168 pin SDRAM", "72 pin EDO",
         "72 pin fast page"};

    // four-param ctor
    public Memory(String ident, double price,
                   int memType, int memSize) {
        id = ident;
        cost = price;
        type = memType;
        size = memSize;
    }
}
public void displayDetails() {
    super.displayDetails();
    System.out.println("Memory type: " +
                       description[type]);
    System.out.println("Memory size: " + size + "MB");
}
Note:

(*) extends in Java: inheritance.
Memory is a subclass (derived class) of PCPart.
Similar to C++s

```java
    class Memory: public PCPart {
        ...
    }
```

(*) The line

```java
    static final int SDRAM = 0, EDO = 1, FASTPAGE = 2;
```

static: only one copy of SDRAM (etc.),
shared by all instances of Memory.
final: it's an error to change SDRAM (etc.).

Similar to C++s

```java
    const int SDRAM = 0, EDO = 1, FASTPAGE = 2;
```
Better C++ would be

```cpp
enum MemoryType {SDRAM, EDO, FASTPAGE};
```

Although Java has enums, they don’t work the same way as in C++. (They’re not “integers with syntactic sugar”.)

So:

```java
Memory m;
...
  m.type = SDRAM;
```

(*) Two methods: ctor, displayDetails()

(*) **Polymorphism**: Memory's displayDetails() method:
  (**) First calls displayDetails() in the superclass PCPart, handling in-common detail display.
  (**) Next, displays stuff particular to Memory objects.

(*) description[] simplifies the code
If PCPart contains the method

```java
public static void main(String args[]) {
    int reduction = 5;  // percentage reduction
    Memory m = new Memory("MEM884", 56,
                           Memory.SDRAM, 128);
    m.displayDetails();
    System.out.println(reduction + "% discount is "
                       + m.discount(reduction));
}
```

then

```
$ java PCPart
Id number is MEM884 Cost is: 56.0
Memory type: 168 pin SDRAM
Memory size: 128MB
5% discount is 2.8
```

What about

```
Memory m = new Memory();
```

Error, since only 4-param ctor!
The MBoard Class

```java
public class MBoard extends PCPart {
    int chipset;    // chipset LX, BX, or TX
    int busSpeed;   // speed of motherboard bus

    // kinds of chipsets, and their descriptions
    static final int LX = 0, BX = 1, TX = 2;
    static final String description[] =
        {"LX", "BX", "TX"};

    // null ctor
    public MBoard() {
    }

    // four-param ctor
    public MBoard(String identifier, double price,
                   int chips, int speed) {
        id = identifier;
        cost = price;
        chipset = chips;
        busSpeed = speed;
    }
}
```
public void displayDetails() {
    super.displayDetails();
    System.out.println("Motherboard chipset used: "+description[chipset] + " Bus speed: " + busSpeed);
}

Note that this class contains a zero-parameter constructor, so we can do
    MBoard mb = new MBoard();
but we'd then need to set the attributes by hand, e.g.,
    mb.id = "LX349";
and so forth.
If PCPart contained

```java
public static void main (String args[]) {
    int reduction = 12;   // percentage reduction
    MBoard mb = new MBoard("LX349", 60.0,
                            MBoard.LX, 100);

    mb.displayDetails();
    System.out.println(reduction +
                       "% discount is " + mb.discount(reduction));
}
```

then

```bash
$ java PCPart
Id number is LX349 Cost is: 60.0
Motherboard chipset used: LX Bus speed: 100
12% discount is 7.2
```

Implementation of Disk and Monitor classes is similar to the Memory and MBoard classes.
Extending a subclass

Pre-built motherboard+processor combination. Create a new MBoardAndProc class, an extension of the MBoard class.

An IS-A relationship: MBoardAndProc IS-A MBoard
```java
class MBoardAndProc extends MBoard {
    int procType;    // PENTIUM, AMDK6, or CYRIX
    int procSpeed;   // MHz

    // processor types
    static final int PENTIUM = 0, AMDK6 = 1, CYRIX = 2;

    // processor descriptions
    static final String description[] = {
        "Pentium II", "AMD K6", "Cyrix mark III"};

    public MBoardAndProc(String identifier,
                          double price, int chips,
                          int speed, int type,
                          int proSpeed) {
        super(identifier, price, chips, speed);
        procType = type;
        procSpeed = proSpeed;
    }
}
```
```java
public void displayDetails() {
    super.displayDetails();
    System.out.println("Processor type: " +
    description[procType] +
    "speed: " + procSpeed);
}

super() in ctor: calls the ctor of superclass.

Test via new main() method:

public static void main (String args[]) {
    int reduction = 12;    // percentage reduction
    MBoardAndProc mbp = new MBoardAndProc("LX349",
        60.0, MBoard.LX, 100,
        MBoardAndProc.PENTIUM, 450);
    mbp.displayDetails();
    System.out.println(reduction +
    "% discount is " +
    mbp.discount(reduction));
}
```
Alternative design: create a Processor class, so that

How used? Later ...
Implementation

```java
public class Processor extends PCPart {
    int procType;
    int procSpeed;
    // processor types
    static final int PENTIUM = 0, AMDK6 = 1, CYRIX = 2;
    // processor descriptions
    static final String description[] = {
        "Pentium II", "AMD K6", "Cyrix mark III"};

    public Processor (String identifier, double price, int type, int proSpeed) {
        id = identifier;
        cost = price;
        procType = type;
        procSpeed = proSpeed;
    }

    public void displayDetails() {
        super.displayDetails();
        System.out.println("Processor type: "+
            description[procType] + "speed: " + procSpeed);
    }
}
```
How to use Mboard, Processor classes?

Try to say that an object of the `MBoardAndProc` class
HAS-A MBoard object
HAS-A Processor object
MboardAndProc doesn't fit into the hierarchy:

![MBoardAndProcessor](image)

We'd then start its implementation as follows:

```cpp
class MBoardAndProc {
    MBoard mb;
    Processor proc;
    ...
}
```
Problem: MBoard and Processor each have
   ID type
   price

No relation between
   price of the combined object
   prices of the components

(Ditto with IDs.)

Avoid HAS-A relation here.
Class and instance variables

```java
public class Mercedes {
    String manufacturer;
    int model;
    double price;
    int engineSize;
    boolean ABS;
    boolean airConditioning;
    boolean CD;

    public static void main(String args[]) {
        Mercedes s320 = new Mercedes();
        Mercedes e430 = new Mercedes();
    }
}
```
All variables are instance variables. Each object has its own copy of every instance variable. Bad idea.

`int` for model? Bad idea: non-mnemonic.

What about `String`? Bad idea:
(*) takes up too much room
(*) how to guard against illegal values ("z–280")?

Better idea?
public class Mercedes {
    static String manufacturer = "Mercedes";
    static final int e200 = 0, e240 = 1, e280 = 2, e320 = 3;
    static final int s280 = 4, s320 = 5, s420 = 6, s500 = 7, s600 = 8;
    static final String description[] = 
    int model;
    double price;    // in dollars
    int engineSize;  // in cc's
    static boolean ABS = true;
    static boolean airConditioning = true;
    boolean CD;
public static void main(String args[]) {
    Mercedes s320 = new Mercedes();
    s320.model = Mercedes.s320;
    s320.price = 28000;
    s320.engineSize = 2996;
    s320.CD = false;
}
Static methods

Methods can be static. A method *must* be static if it changes the value of a class variable.

Non-static method in Mercedes class to change price:

```java
public void priceChange(double change) {
    System.out.print("old price is " + price);
    price += price*change/100.0;
    System.out.println(" new price is " + price);
}
```

It would be invoked as (say):
```
s320.priceChange(5);
```
static method to change the (static) manufacturer:

```java
public static void manufacturerChange(String takeover) {
    System.out.print("old manufacturer is " +
                      manufacturer);
    manufacturer = takeover;
    System.out.println(" new manufacturer is " +
                       manufacturer);
}
```

This would be called as:
```
Mercedes.manufacturerChange("Nissan");
```

Note that in method call:
- class name precedes dot for static method
- object name precedes dot for non-static method
Overloading methods

Identical methods (name, param list) in sub- and superclasses...
   If method exists in the object's class, it's used.
   If not, we check its superclass.
       If it exists, this is used.
   If not, we check its superclass.
       If it exists, this is used. (etc.)

Can have identically-named methods with non-identical parameter lists in same class.
   Java checks name, param list for a match.
   Example: System.out.println() can take
            a string, a double, an int ...
            as its parameter
   This is called overloading.

Unlike C++, no overloaded operators in Java.
Example:

```java
public class Foo {
    public static void main(String[] args) {
        Foo f = new Foo();
        f.bar();
        f.bar("hello");
        f.bar(17, 22);
    }

    public void bar() {
        System.out.println("hello, world");
    }

    public void bar(String s) {
        System.out.println(s);
    }

    public void bar(int x, int y) {
        System.out.println(x + " + " + y + " = " + (x + y));
    }
}
```
$ java Foo
hello, world
hello
17 + 22 = 39
Overloading and ctors ...

ctor can have
  zero parameter version
  non-zero parameter versions

If you don't write any ctors, Java supplies a default zero-parameter ctor (essentially does nothing).

If you provide *any* ctors, then the default zero-parameter ctor is *not* created, and you must define your own.
Access specifiers

public:
  access allowed by all methods in all packages
protected:
  access allowed by all methods in the class
  or in any subclass
private:
  access allowed only by methods
  within the class itself
package: (default)
  access allowed by classes and subclasses
  in same package

Precede name of entity's type:

    public void foo() { ... }  
    private int opCode;
    protected float porosity;

(Compare with C++.)
Outer classes: Foo.java might contain

    public class Foo { ... }
    class Bar { ... }

Inner classes: Foo.java might contain

    public class Foo {
        ... stuff ...
        private class Bar { ... }
            ... more stuff ...
    }
CLASSPATH

... where Java looks for classes (env var)
Some useful classes will be kept in
   ~agw/java-classes
This means that you should put
       setenv CLASSPATH .:~agw/java-classes
in your ~/.cshrc file.

If you develop a library of useful classes, you may want to put same in a
specific directory, e.g., ~/java-classes, in which case you would want
       setenv CLASSPATH .:~/java-classes:~agw/java-classes
in your ~/.cshrc file.

If you use the bash shell, you should put
       export CLASSPATH=.:~agw/java-classes
(etc.) in your ~/.bashrc file.
Packages

... collection of related classes.
Import a package *whatever* by
    import *whatever*;
e.g.,
    import java.io.OutputStream;
    import java.io.*;
Canonical order:
    general->specific (opposite of DNS)

To make a class part of a package, put
    package *whatever*;
at beginning of file containing class, e.g.,
    package edu.fordham.dsm.scheduler;

Packaging a package :-)

(*) set of class files (*Foo.class*, *Bar.class*, ...)
    Put them in directory
        ~/java-classes/edu/fordham/dsm/scheduler
(*) jar (Java ARchive) file (*scheduler.jar*)
    Put *scheduler.jar* in directory
        ~/java-classes/edu/fordham/dsm
Interfaces

C++: multiple inheritance
Java: no multiple inheritance
workaround?

interface: series of defns of abstract methods
class inherits an interface via implements keyword
Example: A ball hierarchy:

Inflatable:
  - football, soccer ball, basketball
Kickable:
  - soccer ball, football
Neither inflatable nor kickable:
  - baseball
Inflatable, but not kickable:
  - basketball
(Kickable, but not inflatable?)
Partial implementation:

Inflatable.java:

    public interface Inflatable {
        public void inflate();
    }

Kickable.java:

    public interface Kickable {
        public void kick();
    }

Sportsball.java:

    public class SportsBall {
        public void manufacture() {
            System.out.println("Making a sports ball.");
        }
    }
SoccerBall.java:

```java
public class SoccerBall extends SportsBall implements Inflatable, Kickable {
    public void inflate() {
        System.out.println("Inflating a soccer ball.");
    }
    public void kick() {
        System.out.println("Kicking a soccer ball.");
    }
    public static void g1(Inflatable x) {
        x.inflate();
    }
    public static void g2(Kickable y) {
        y.kick();
    }
    public static void g3(SportsBall z) {
        z.manufacture();
    }
}
```
public static void main(String args[]) {
    SoccerBall ball = new SoccerBall();
    ball.inflate();
    ball.kick();
    ball.manufacture();

    g1(ball);
    g2(ball);
    g3(ball);
}

When run:

$ java SoccerBall
Inflating a soccer ball.
Kicking a soccer ball.
Making a sports ball.
Inflating a soccer ball.
Kicking a soccer ball.
Making a sports ball.