What we'll cover today and Friday

- Classes and subclasses
- Abstract classes and abstract methods
- Polymorphism: what is it and how does it work?
- Interfaces
- Inheritance vs. composition

Example: A client-side streaming media measurement program

Problem statement

- Different types of media players
  - Windows Media Player, RealOne, QuickTime, mplayer, ...
- Different OSes
  - Linux/Unix, Windows 95/98/NT/2000/XP, Mac, ...
- Q: How do we write software to do this, without having to rewrite it for every different platform and media player??
Thinking about the solution

What do all media players have in common?

Thinking about the solution

- We can use the concept of inheritance to address this problem
- Advantages:
  - reuse code
  - define common data and methods once
  - easier to extend, maintain
- We can solve this problem in several different ways, using different methods of inheritance

Solution #1: Use subclasses

- A subclass is an extension of another class
  - inherits all protected and public data and methods from its parent class
  - private data and methods are hidden from a subclass
- Parent class is referred to as the superclass of this object.

Defining subclasses in Java

Parent class:
```java
class Parent {...}
```
Child class:
```java
class ChildOfParent extends Parent {...}
```
Note: A Java class can only have one superclass (more on this later)
- C++ classes can have multiple parents
Back to our example: superclass

class MediaPlayer {
    /* Variables and constants */
    public static final int AUDIO = 0;
    public static final int AUDIO_VIDEO = 1;
    protected String streamName;
    protected Vector fileHistory;
    protected int fileType;
    protected int duration;
    protected Server mediaServer;
    /* Constructor */
    public MediaPlayer(String streamName) {
        this.streamName = streamName;
        ...
    }
    /* continued on next slide */
}

Code continued from previous slide

    /* Methods */
    public boolean play() {
        duration = mediaServer.getPlayTime(stream);
        type = mediaServer.getStreamType(stream);
        ...
    }
    public boolean stop() {
        super.stop();
        /* WMP-specific instructions go here */
    }
    public boolean fastForward() {
        ...
    }
    public boolean rewind() {
        ...
    }

Subclass #1

class WindowsMediaPlayer extends MediaPlayer {
    public WindowsMediaPlayer(String stream) {
        super(stream);
        /* additional initialization tasks go here */
    }
    public boolean play() {
        /* WMP-specific instructions go here */
    }
    public boolean stop() {
        super.stop();
        /* WMP-specific instructions go here */
    }
    ...
}

Subclass #2

class RealOnePlayer extends MediaPlayer {
    /* Constructor */
    public RealOnePlayer(String stream) {
        super(stream);
        /* additional initialization tasks go here */
    }
    public boolean play() {
        /* RealOne-specific instructions go here */
    }
    public boolean stop() {
        super.stop();
        /* RealOne-specific instructions go here */
    }
    ...
}
Notes

- Subclasses always call their superclass' constructor (`super()`)  
  - will be done automatically if we don't include it  
  - so, if the superclass' constructor expects input arguments, we must call `super(<args>)` explicitly  
  - **must** be the first statement!

- A superclass' methods and variables may be overriden by the subclasses  
  - to call the method in the superclass, use `super.methodName(...)`

The main program

```java
class ClientSystemMain {
    public static void main (String[] args) {
        WindowsMediaPlayer wmp;  
        RealOnePlayer rop;  
        /* Get the type of media player to use from the command line. This is an integer value */  
        int playerType = Integer.parseInt(args[0]);  
        /* Get the stream name from the command line*/  
        String stream = args[1];  
        /* Initialize the player */  
        switch(playerType) {
            case 1:  /* windows media player */  
                wmp = new WindowsMediaPlayer(stream);  
                break;  
            case 2:  /* real one player */  
                rop = new RealOnePlayer(stream);  
                break;
        }/* Code continues on the next page */
    }
    /* Play the specified stream */
    switch (playerType) {
        case 1:  /* windows media player */
            wmp.play();
            break;
        case 2:  /* real one player */
            rop.play();
            break;
    }
    /* Stop the specified stream */
    switch (playerType) {
        case 1:  /* windows media player */
            wmp.stop();
            break;
        case 2:  /* real one player */
            rop.stop();
            break;
    }
    /* Code continues on the next page */
}
```

What's the problem with this approach?

- Awkward!
- Have to maintain `switch` statements for every possible media player option  
  - what if we have 20 different subclasses of media players?
- Programming rule of thumb: The more lines of code, the more chances you have for bugs
- Q: Can we simplify this somehow?
Answer: Polymorphism

- **Polymorphism** allows us to use a single variable to refer to objects of different (related) classes.

- **Idea:** Declare a variable that corresponds to the superclass, and *downcast* it on the fly.
  - decide at runtime what the variable actually refers to
  - no need to pre-anticipate every possible case

- **Advantages:**
  - streamlines and simplifies the code
  - allows us to add classes to the hierarchy later without having to change the code that calls the class

Redo the main program using polymorphism

```java
class ClientSystemMain {
    public static void main (String[] args) {
        MediaPlayer mp;
        /* Get the type of media player to use from the command line. This is an integer value. */
        int playerType = Integer.parseInt(args[0]);
        /* Get the stream name from the command line*/
        String stream = args[1];
        /* Initialize the player */
        /* Unfortunately, we still have to include the switch statement here. */
        ...
        /* Play the specified stream */
        mp.play();
        /* Stop the specified stream */
        mp.stop();
    }
}
```

Back to the original problem

- We solved the original problem using subclasses.
- Disadvantage: we have to *define* common methods, data, etc. in the superclass.
- Q: What if we want to leave all or some of the implementation details to the subclasses?

Solution #2: Abstract classes

- An *abstract class* is a class that cannot be instantiated.
- Any class that contains an *abstract method* is an abstract class.
- An *abstract method* is a method which is not defined in a class.

```java
abstract public boolean defineThisLater();
```

- **private and static methods cannot be declared** abstract.
Redefine `MediaPlayer` as an abstract class

```java
abstract class MediaPlayer {
    /* Variables and constants */
    public static final int AUDIO = 0;
    public static final int AUDIO_VIDEO = 1;
    protected String streamName;
    protected Vector fileHistory;
    protected int fileType;
    protected int duration;
    protected Server mediaServer;
    /* Constructor */
    public MediaPlayer(String streamName) {
        this.streamName = streamName;
        ...
    }
    /* continued on next slide */
}
```

Q: If a class is abstract, how can it have a constructor?

- Abstract classes can define variables
  - constructor can initialize these variables
- Abstract classes can have fully-defined methods
  - constructor can call these methods
- Most likely, we will want to perform some common tasks when creating instances of the subclasses
  - constructor in abstract superclass takes care of this for us (it's called automatically anyway)

Code continued from previous slide

```java
/* Methods */
abstract public boolean play();
abstract public boolean stop();
abstract public boolean fastForward();
abstract public boolean rewind();
}
```

Notes

- Define the subclasses the same way as in Solution #1
- All abstract methods must end with a semi-colon
  - no curly braces!
- The subclasses will still inherit the variables from `MediaPlayer`, along with any non-abstract methods
- The subclasses now **must** define the abstract methods!
- Can still use polymorphism in the main program to refer to the subclasses of `MediaPlayer`
import java.awt.*;
public class Shape {
    protected double width, height, x, y;
    protected Color color;
    public Shape(double x, double y, double w, double h, Color c) {
        this.x=x; this.y=y;
        width = w; height = h;
        color = c;
    }
    public double findArea() {
        return width * height;
    }
    public double findPerimeter() {
        return (2*width + 2*height);
    }
}

import java.awt.*;
public class Rectangle extends Shape {
    public Rectangle(double x, double y, double w, double h, Color c) {
        super(x,y,w,h,c);
    }
}

import java.awt.*;
public class Circle extends Shape {
    protected double radius;
    public Circle(double x, double y, double r, Color c) {
        super(x,y,2*r,2*r,c);
        radius = r;
    }
    public double findArea() {
        return Math.PI*radius*radius;
    }
    public double findPerimeter() {
        return 2*Math.PI*radius;
    }
}

import java.awt.*;
public class TestShapes {
    public static void main(String[] args) {
        Shape[] shapeList = new Shape[4];
        shapeList[0] = new Circle(34, 48, 10, Color.red);
        shapeList[1] = new Rectangle(100, 200, 50, 50, Color.blue);
        shapeList[2] = new Rectangle(45, 45, 100, 230, Color.red);
        shapeList[3] = new Circle(50, 100, 5, Color.green);
        for (int i=0; i<shapeList.length; i++) {
            System.out.println("Area of shape "+i+" is "+shapeList[i].findArea());
            System.out.println("Perimeter of shape "+i+" is "+shapeList[i].findPerimeter());
        }
    }
}
Another polymorphism example

```java
import java.awt.*;
public class TestShapes {
    public static void main(String[] args) {
        Shape[] shapeList = new Shape[4];
        shapeList[0] = new Rectangle(34, 48, 10, 20, Color.yellow);
        shapeList[1] = new Rectangle(100, 200, 50, 50, Color.blue);
        shapeList[2] = new Rectangle(45, 45, 100, 230, Color.green);
        shapeList[3] = new Circle(50, 100, 5, Color.green);
        for (int i=0; i<shapeList.length; i++) {
            System.out.println("Area of shape "+i+" is "+shapeList[i].findArea());
            System.out.println("Perimeter of shape "+i+" is "+shapeList[i].findPerimeter());
        }
    }
}
```

Shape as an abstract class

```java
import java.awt.*;
abstract public class Shape {
    protected double width, height, x, y;
    protected Color color;
    public Shape(double x, double y, double w, double h, Color c) {
        this.x=x; this.y=y;
        width = w; height = h;
        color = c;
    }
    abstract public double findArea();
    public double findPerimeter() {
        return (2*width + 2*height);
    }
}
```

Rectangle inheriting from abstract Shape class

```java
import java.awt.*;
public class Rectangle extends Shape {
    public Rectangle(double x, double y, double w, double h, Color c) {
        super(x,y,w,h,c);
    }
    public double findArea() {
        return width*height;
    }
}
```

Calculating area

- Area can be calculated for a variety of objects
  - 2-d shapes
  - 3-d shapes (surface area)
  - curves
  - plots of land
  - ...
- These objects have a few similarities, but not many
One idea

- Make one superclass from which all of these objects inherit
- What are the problems with this approach?

A better solution

- These objects have similar behavior without having similar traits
- If we could just define this behavior, and have the classes inherit just this instead, that would be ideal
  - behavior = action = method
- Can we have just a set of methods, and not a full class?

Answer: yes!

- An interface is a collection of method definitions
- The methods are defined but **not implemented**
- Interfaces may contain constants, but **no other variables**
- Interfaces define behaviors that more than one class may want to implement
- Examples of Java interfaces: `ActionListener`, `MouseListener`
- Indicated by the `implements` keyword

What's the difference between an interface and an abstract class?

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Excerpted from *The Java Tutorial*, Track Learning the Java Language, Lesson Interfaces and Packages, section What is an Interface?
Shape example, using an interface

public interface AreaCalculator {
    public double findArea();
}

The Circle class, using an interface

import java.awt.*;
public class Circle implements AreaCalculator{
    protected double radius, x, y;
    protected Color color;
    public Circle(double x, double y, double r, Color c) {
        this.x=x; this.y=y;
        radius = r;
        color = c;
    }
    public double findArea() {
        return Math.PI*radius*radius;
    }
}

The Rectangle class, using an interface

import java.awt.*;
public class Rectangle implements AreaCalculator {
    protected double width, height, x, y;
    protected Color color;
    public Rectangle(double x, double y, double w, double h, Color c) {
        this.x=x; this.y=y;
        width = w; height = h;
        color = c;
    }
    public double findArea() {
        return width * height;
    }
}

Another class using the same interface

import java.awt.*;
public class Cylinder implements AreaCalculator{
    protected double radius, x, y, height;
    protected Color color;
    public Cylinder(double x, double y, double r, double h, Color c) {
        this.x=x; this.y=y;
        radius = r; height = h;
        color = c;
    }
    public double findArea() {
        // area = surface area
        return (2*Math.PI*radius*radius +
                2*Math.PI*radius*height);
    }
}
Can use polymorphism with interfaces, too!

```java
import java.awt.*;
public class Areas {
    public static void main(String[] args) {
        AreaCalculator[] a = new AreaCalculator[3];
        a[0] = new Circle(34, 48, 10, Color.red);
        a[1] = new Rectangle(100, 200, 50, 50, Color.blue);
        a[2] = new Cylinder(300, 20, 10, 50, Color.green);
        for (int i=0; i<a.length; i++) {
            System.out.println("Area of item "+i+" is "+a.findArea());
        }
    }
}
```

Back to our original example

![Diagram](image-url)

Limitations of inheritance

- Sometimes, inheritance is not a good idea
- Example: Both DataCollector and ExternalCommunicator communicate using datagram sockets (class DatagramSocket in Java)
- Idea: Let both classes inherit from DatagramSocket
  - this is a bad idea!

If ExternalCommunicator was a subclass of DatagramSocket...

- Most of the functionality of DatagramSocket is not used by ExternalCommunicator
  - in fact, we only use receive()
  - lots of wasted resources!
- A better solution: Use composition
Composition

- Composition means that one class contains an object from another class
- Use composition when you only want to use the public methods/data of a class, but do not want your class to relate to it
- We’ve used composition many times in this class
  - SketchPadFrame contains a DrawPanel
  - MiniYahtzee contains a Die
  - ExternalCommunicator contains a DatagramSocket

When to use (and not to use) inheritance: Summary

- Use subclasses (with a superclass or abstract superclass) when two or more objects share similar data and methods
  - “is-a” relationship
- Examples:
  - a Rectangle is a Shape
  - a Circle is a Shape

When to use (and not to use) inheritance: Summary (cont)

- Use interfaces when you have a behavior that dissimilar classes share in common
  - “uses-a” relationship
- Examples:
  - SketchPadFrame uses an ActionListener
  - Cylinder uses an AreaCalculator

When to use (and not to use) inheritance: Summary (cont)

- Use composition when one object contains another (dissimilar) object
  - “has-a” relationship
- Examples:
  - MiniYahtzee has a Die
  - SketchPadFrame has a DrawPanel
  - SketchPadFrame has a JButton
Abstract superclass vs. fully-defined superclass

- Choice depends on the situation
- Questions to ask:
  - which approach allows you to modify and extend the class more easily?
  - does your design allow for future extensions and modifications?

Additional reading

- Interfaces
  - Java tutorial: Lesson on *Interfaces and Packages*
- Inheritance (if you don’t have Wu’s book)
  - Java tutorial: Lesson on *Managing Inheritance*
  - http://java.sun.com/docs/books/tutorial/java/javaOO/subclasses.html