What we'll cover today

- 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?

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

```java
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

```java
/* Methods */
public boolean play() {
    duration = mediaServer.getPlayTime(stream);
    type = mediaServer.getStreamType(stream);
    ...
}
public boolean stop() {
    ...
}
public boolean fastForward() {
    ...
}
public boolean rewind() {
    ...
}
```

Subclass #1

```java
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() {
        /* WMP-specific instructions go here */
    }
    ...
}
```

Subclass #2

```java
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() {
        /* RealOne-specific instructions go here */
    }
    ...
}
```
Notes

- Subclasses always call their superclass' constructor (`super()`)
- A superclass' methods and variables may be overridden by the subclasses

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]);
      /* Set up the components of the client-side system */
      ...
      /* Initialize the player */
      ...
      /* Code continues on the next page */
   }
}
```

The main program (continued)

```java
   /* 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;
   }
```

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]);
    /* Set up the components of the client-side system */
    ...
    /* Initialize the player */
    ...
    /* 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

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 */
}

Code continued from previous slide

/* 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.
- The subclasses will still inherit the variables from MediaPlayer, along with any non-abstract methods
- The subclasses now **must** define the abstract classes!
- Can still use polymorphism in the main program to refer to the subclasses of MediaPlayer

Back to our original example

![Diagram showing data flow between MediaPlayer, Data Collector, and External Communicator]

- Media player
- Data collector
- External Communicator
- From server
- To server
Problem statement

- The data collector and communicator need to maintain lists
  - communicator: access control list (who can I communicate with?)
  - data collector: list of streams for which it has collected data in the past 24 hours (and links to where this data is stored)
- How can we do this?
  - Write the same or similar functions in each component?
    - repetition is bad!
  - Have each component inherit from the same superclass?
    - what if they are not similar, as in this case?

Answer: interface

- 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: Runnable, EventListener, Collection

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.

Solution using interfaces

- ListHandler.java
- VectorList.java
- HashList.java

Excerpted from The Java Tutorial, Track Learning the Java Language, Lesson Interfaces and Packages, section What is an Interface?
The main code using interfaces

```java
class ExternalCommunicator {
  ...
  /** The access control list: who can contact this client */
  private ListHandler acl;
  ...
  /** Constructor */
  public ExternalCommunicator(...) {
    acl = new HashList();
    /* Add a host to the list */
    acl.addEntry(filename, streamName);
  }
  /** Receive and process an incoming message on the socket
  * @param sock The socket on which the message arrived
  */
  /* continued on next page */
}
```

Limitations of inheritance

- Sometimes, inheritance is not a good idea
- Example: ExternalCommunicator contains a DatagramSocket. Why didn't we make ExternalCommunicator a subclass of DatagramSocket?

If ExternalCommunicator was a subclass of DatagramSocket...

- Most of the functionality of DatagramSocket is not used by ExternalCommunicator
- We only use receive()
  - Lots of wasted resources!
- A better solution: Use composition

Code continued from previous page

```java
private void processMessage(DatagramSocket sock) {
  DatagramPacket packet = new DatagramPacket(...);
  sock.receive(packet);
  /* check to see if packet was received from a valid address */
  if (acl.isValid(packet.getAddress())) {
    ...
  } else {
    System.err.println("Invalid sender");
  }
  ...
}
```
### 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.
- `ExternalCommunicator` utilizes composition.

### 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
- Use *interfaces* when you have a behavior that dissimilar classes share in common.
  - "uses-a" relationship
- Use *composition* when one object contains another (dissimilar) object.
  - "has-a" relationship

### Abstract superclass vs. fully-defined superclass

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

### Additional reading

- **Interfaces**
  - Java tutorial: Lesson on *Interfaces and Packages*  
    - [http://java.sun.com/docs/books/tutorial/java/interpack/interfaces.html](http://java.sun.com/docs/books/tutorial/java/interpack/interfaces.html)
- **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](http://java.sun.com/docs/books/tutorial/java/javaOO/subclasses.html)