**Decorator Pattern**

* goal: create an object to represent a song "log"(i.e., record of what songs were played)
  + provide ability to answer queries about songs played
  + don't want to create a subclass
* Functionality:
  + void recordInLog(Song s)
  + void recordInLog(Song s, Date time)
  + Date lastPlayed(Song s)
* Java's class for dealing with Date is called "Date"
  + separate classes for formatting and calendar:
    - Calendar
    - DateFormat
    - SimpleDateFormat
* You COULD use two arraylists, one for the song and one for the date, and getters and setters for these two fields
  + but this isn't the most elegant way b/c the two arrays are tied to each other logically, but not in code
* Decorator pattern is useful when you want to take an existing class and add additional state or funcitonality to a class without formally sub classing
  + usually, sub classing is not ALWAYS needed, so sub classing is not preferable\
  + When use decorator?
    - additional state/functionality is auxiliary to object's main purpose
    - additional state/functionality is local to a collection or some other class encapsulating the object
      * Songlog for example
    - as a way of emulating multiple inheritance
      * want to build up an object with subsets of different functionality
* Decorator pattern relies on delegation

* Setting up decorator:
  + start with original interface
    - if you're decorating a class without an interface, refactor the original class to have an interface
  + Step 1: extend interface, declaring additional functionality
    - in song example, this is LoggedSong that has a Date getDate();
  + Step 2: create a class that implements decorated interface
    - constructor is given an instance of the original, undecorated type
      * Ex:

public LoggedSongImpl(Song s, Date d){ song = s; date = d;};

* may also need to provide other parameters that are required for the additional functionality you're trying to add(Date object)
* then, delegate the original interface methods to the encapsulated original object

Ex:

public boolean equals(Song s){ return song.equals(s);}

* provide implementations for additional decorated behavior
* Now that you have a decorator set up, you just have to make ONE arraylist object that is of type LoggedSong
  + then when you record new song objects into the log, you just use the decorator interface
  + Ex:

public void recordInLog(Song s, Date d){

logged\_song\_list.add(new LoggedSongImpl(s,d);

}

* A decorated interface is awesome because old code that uses un-decorated interfaces will not have to change, because the decorated interfaces contain an un-decorated object that still works exactly the same

* the big thing is that the new decorated implementation has no relation to the original implementation
  + only the interfaces of each implementation are related(decorated interface extends original interface)
* Unwrapping the decorator
  + sometimes the place where you're using decorator in order to add additional functionality, it might be necessary to "undecorate" or "unwrap" the original object
    - for example, if we need to give it back to someone who expects the original object
    - this is pretty easy, just create a getter method for the original object
* decorator comes in handy when you're maintaining a collection that is useful for only a particular abstraction