* Java Iterator Pattern Interfaces
  + java provides two generic interfaces for supporting the iterable design pattern
    - implemented by various collection types such as List<E>, Map<E>, Set<E>, etc.
      * all of these collection types are of type Iterable<E>
      * the Iterator<E> has the following methods:
        + boolean hasNext(), E next(), void remove()
      * Ex:

Iterator<Song> iter = slist.iterator();

while(iter.hasNext()){

Song nextSong = iter.next();

System.out.println("The next song is " + nextSong.getName());

}

* Simultaneous iterator - two iterators, one list
  + when one iterator advances, the other one remains still unless you call it to advance
  + same list, two different traversals
* for-each
  + a loop that does the job of an iterator that traverses a list
  + Ex:

for(E elem: e\_list){

//do something to elem

}

* you can also subclass Iterator<E> that adds functionality
  + for example, the iterator subclass can override the method 'boolean hasNext()' and check for something in a list's element
  + Ex: hasNext() can check if a song has a minimum rating
* Factory Design Pattern
  + used when direct construction is not always good(using the "new" keyword)
  + several different contexts when factory design pattern is appropriate
    - Singleton
      * when there should only be one instance of a particular class in the whole system
      * generally the object represents some sort of system-wide resource that may be needed by many objects in different parts of the software
        + stored in a private static variable and retrieved by a static getter(the factory method)
    - Multiton
      * when there should be only one instance of an object associated with some identifying property
      * one object for every identifying trait
        + request one student using a PID
      * look for existing object to match your need; if no, then create one to match your need
        + usually implemented using a map

maps don't take in int or char, but rather Integer and Character

* + - Dynamic subclass binding
      * when the specific subclass of an object can only be determined at the time that it is created
      * a superclass could check a boolean field to either create one subclass or another
        + if(online){return new OnlineService();}

else{return new OfflineService();}

* complex construction
  + when the construction of the object requires complex validation and/or side effects that must be handled
  + when null should be a valid result
  + when an existing object might need to be provided instead of a new one

* How to: Factory Pattern
  + make the constructor private
    - prevents direct use of the 'new' keyword for creating new objects outside of the class
    - provide a static 'factory' method that constructs new instances by calling the constructor and then returns them