HW8

PlayList

```
 **/Represnts a list of musical tracks. The list has a maximum capacity (int),
  * and an actual size (number of tracks in the list, an int)/* .
class PlayList}
    private Track[] tracks;  // Array of tracks (Track objects)
    private int maxSize;     // Maximum number of tracks in the array
    private int size;        // Actual number of tracks in the array

 **/     Constructs an empty play list with a maximum number of tracks /* .
    public PlayList(int maxSize)}
        this.maxSize = maxSize;
        tracks = new Track[maxSize];
        size = 0;
{

 **/     Returns the maximum size of this play list /* .
    public int getMaxSize} ()
        return maxSize;
{

 **/     Returns the current number of tracks in this play list /* .
    public int getSize} ()
        return size;
{

 **/     Method to get a track by index/*
    public Track getTrack(int index)}
        if (index >= 0 && index < size)}
            return tracks[index];
 {        else}
```

```
            return null;
{
{

 **/    Appends the given track to the end of this list .
  *     If the list is full, does nothing and returns false.
  *     Otherwise, appends the track and returns true/* .
    public boolean add(Track track)}
        if(getSize() == getMaxSize() + 1)}
            return false;
{
        else}
            tracks[getSize()] = track;
            size;++
{
        return true;
{

 **/    Returns the data of this list, as a string. Each track appears in a separate line/* .
 ////    For an efficient implementation, use StringBuilder.
    public String toString} ()
        StringBuilder sb = new StringBuilder;()
        for(int i = 0;i<getSize();i++)}
            sb.append(tracks[i].toString());
            sb.append("\n");
{
        return sb.toString;()
{

 **/    Removes the last track from this list. If the list is empty, does nothing/* .
     public void removeLast} ()
```

```
        if(getSize()!=0)}
            tracks[getSize()-1] = null;
            size;--
{
{

 **/    Returns the total duration (in seconds) of all the tracks in this list/*.
    public int totalDuration} ()
        int sumDuration = 0;
        for(int i = 0; i<getSize(); i++)}
            sumDuration += tracks[i].getDuration;()
{
        return sumDuration;
{

 **/    Returns the index of the track with the given title in this list.
  *     If such a track is not found, returns -1/* .
    public int indexOf(String title)}
        for(int i = 0; i<getSize(); i++)}
            if(tracks[i].getTitle()==title)}
                return i;
{
{
        return -1;
{

 **/    Inserts the given track in index i of this list. For example, if the list is
  *     (t5, t3, t1), then just after add(1,t4) the list becomes (t5, t4, t3, t1).
  *     If the list is the empty list (), then just after add(0,t3) it becomes (t3).
  *     If i is negative or greater than the size of this list, or if the list
  *     is full, does nothing and returns false. Otherwise, inserts the track and
```

```
 *      returns true/* .
    public boolean add(int i, Track track)}
        if (getSize() >= getMaxSize() || i < 0 || i > getSize())}
            return false;
{
        for (int j = getSize() - 1; j >= i; j--)}
            tracks[j + 1] = tracks[j];
{
        tracks[i] = track;
        size;++
        return true;
{



 **/    Removes the track in the given index from this list.
  *     If the list is empty, or the given index is negative or too big for this list ,
  *     does nothing and returns -1/* .
    public int remove(int i)}
       if (getSize() == 0 || i < 0 || i > getSize())}
           return -1;
{
           for (int j = i; j < getSize() ; j++)}
            tracks[j] = tracks[j+1];
{
        size;--
        return 0;
{


 **/    Removes the first track that has the given title from this list.
  *     If such a track is not found, or the list is empty, or the given index
  *     is negative or too big for this list, does nothing/* .
```

```
    public void remove(String title)}
        for (int i = 0; i < getSize() ; i++)}
            if(tracks[i].getTitle() == title)}
                remove(i);
{
{
{

 **/    Removes the first track from this list. If the list is empty, does nothing/* .
    public void removeFirst} ()
        remove;(0)
{

 **/    Adds all the tracks in the other list to the end of this list .
  *     If the total size of both lists is too large, does nothing/* .
 ////    An elegant and terribly inefficient implementation.
     public void add(PlayList other)}
        if (other.getSize() + getSize()<=getMaxSize())}
            for(int i = 0; i<other.getSize();i++)}
                add(other.getTrack(i));
{
{
{

 **/    Returns the index in this list of the track that has the shortest duration,
  *     starting the search in location start. For example, if the durations are
 ,7 ,8 ,5 ,7 ,6 ,1 ,7  *     then min(2) returns 4, since this the index of the
  *     minimum value (5) when starting the search from index 2  .
  *     If start is negative or greater than size - 1, returns -1.
/*
    private int minIndex(int start)}
```

```
        int minDuration = tracks[start].getDuration;()
        int place = start;
        if(start<0 || start>getSize())}
            return -1;
{
        for (int i = start; i<getSize(); i++)}
            if(tracks[i].getDuration()<minDuration)}
                minDuration = tracks[i].getDuration;()
                place = i;
{
{
        return place;
{

 **/    Returns the title of the shortest track in this list .
  *     If the list is empty, returns null/* .
    public String titleOfShortestTrack} ()
        return tracks[minIndex(0)].getTitle;()
{

 **/    Sorts this list by increasing duration order: Tracks with shorter
  *     durations will appear first. The sort is done in-place. In other words,
  *     rather than returning a new, sorted playlist, the method sorts
  *     the list on which it was called (this list)/* .
    public void sortedInPlace} ()
 //         Uses the selection sort algorithm  ,
 //         calling the minIndex method in each iteration.
         for (int i = 0; i < getSize() - 1; i++)}
            int minIndex = minIndex(i);
            Track temp = tracks[i];
            tracks[i] = tracks[minIndex];
```

```
            tracks[minIndex] = temp;
{

{
{
```

Track

```
/** Represents a music track. A track has a title (String), an artist (String),
 *  and a duration (int), in seconds. */
class Track {
   private String title;
   private String artist;
   private int duration;

   /** Constructs a track from the given values. */
   public Track(String title, String artist, int duration) {
      this.title = title;
      this.artist = artist;
      this.duration = duration;
   }

   /** Returns this track's data as "artist, title, minutes:seconds".
    *  For example, "John Lennon, Imagine, 3:07" */
   public String toString() {
      int minutes = (int)(duration/60);
      int seconds = duration%60;
      String newSeconds = String.valueOf(seconds);
      if(seconds<10){
         newSeconds = "0" + String.valueOf(seconds);
      }
      return artist + ", " + title + ", " + minutes + ":" + newSeconds;
```

```
}

/** Returns this track's title. */
public String getTitle() {
   return title;
}
/** Returns this track's artist. */
public String getArtist() {
   return artist;
}
/** Returns this track's duration. */
public int getDuration() {
   return duration;
}

/** If this track's duration is shorter than the other track's duration
 *  returns true; otherwise returns false. */
public boolean isShorterThan(Track other) {
   return duration < other.duration;
}

// Returns a string that represents the totalSeconds as "minutes:seconds",
// Where seconds is always two digits. For example, "3:17" or "12:05".
private String formattedDuration(int totalSeconds) {
   int minutes = totalSeconds/60;
   int seconds = totalSeconds%60;
   String newSeconds = String.valueOf(seconds);
   if(seconds<10){
      newSeconds = "0" + String.valueOf(seconds);
   }
```

```
        return minutes + ":" + newSeconds;
    }
}
```