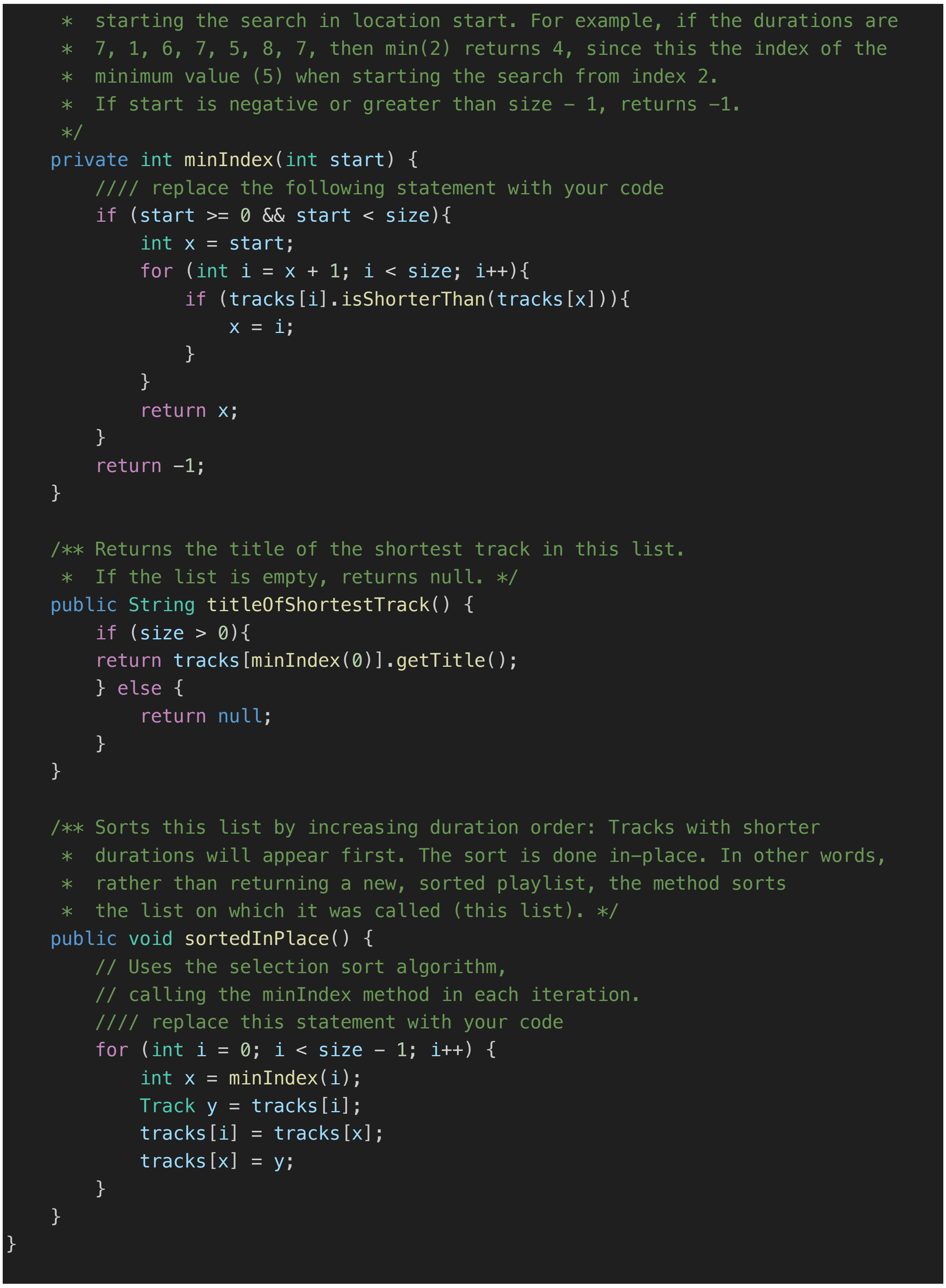
PlayList.java

```java
/** 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) {
        //// replace the following statement with your code
        if (size < maxSize) {
            tracks[size++] = track;
            return true;
        }
        return false;
    }

    /** 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() {
        //// replace the following statement with your code
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < size; i++) {
            result.append(tracks[i].toString()).append("\n");
        }

        return result.toString();
    }

    /** Removes the last track from this list. If the list is empty, does nothing.
*/
     public void removeLast() {
        //// replace this comment with your code
        size--;
    }

    /** Returns the total duration (in seconds) of all the tracks in this list.*/
    public int totalDuration() {
        //// replace the following statement with your code
        int total = 0;
        for (int i = 0; i < size; i++) {
            total += tracks[i].getDuration();
        }
        return total;
    }

    /** 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 < size; i++) {
            if (tracks[i].getTitle().equals(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) {
        //// replace the following statement with your code
        if (i >= 0 && i <= size && size < maxSize) {
            for (int j = size; j > i; j--) {
                tracks[j] = tracks[j - 1];
            }
            tracks[i] = track;
            size++;
            return true;
        }
        return false;
    }

    /** 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 void remove(int i) {
        //// replace this comment with your code
        if (i >= 0 && i < size) {
            for (int j = i; j < size - 1; j++) {
                tracks[j] = tracks[j + 1];
            }
            size--;
        }
    }

    /** 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) {
        //// replace this comment with your code
        int x = indexOf(title);
        if (x != -1){
            remove(x);;
        }
    }

    /** Removes the first track from this list. If the list is empty, does nothing.
*/
    public void removeFirst() {
        //// replace this comment with your code
        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) {
        //// replace this comment with your code
        for (int i = 0; i < other.size; i++) {
            add(other.getTrack(i));
        }
    }

    /** Returns the index in this list of the track that has the shortest duration,
```



```java
     *  starting the search in location start. For example, if the durations are
     *  7, 1, 6, 7, 5, 8, 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) {
        //// replace the following statement with your code
        if (start >= 0 && start < size){
            int x = start;
            for (int i = x + 1; i < size; i++){
                if (tracks[i].isShorterThan(tracks[x])){
                    x = i;
                }
            }
            return x;
        }
        return -1;
    }

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

    /** 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.
        //// replace this statement with your code
        for (int i = 0; i < size - 1; i++) {
            int x = minIndex(i);
            Track y = tracks[i];
            tracks[i] = tracks[x];
            tracks[x] = y;
        }
    }
}
```