Please note that this document is a requirements specification and not a template for the classes you will come up with. You should think about the problem and come up with a design that best suits the specified requirements. First do the design using UML and get it approved before proceeding to implement it. This will save a lot of backtracking and wasted effort (you might have realized from project 2).

You will be using the Date and Time classes (both provided; please study them for coding guidelines, javadoc comments, correct use and encapsulation of classes, attribute and method specifications) as utility in this project for dealing with date of birth (dob), start date, end date, time of event, duration etc. You will be using a constructor, and other methods including the compareTo() method from each class. Please study them.

Your project will utilize all the classes and interfaces (designed and provided) from Project 4. You will be adding new classes related to GUI. You will use the classes developed in earlier projects.

General description of the problem (applicable to all projects)

A. **The enterprise class (StadiumMgmt) – concrete class**
B. **Employee interface**
C. **Person as abstract class**
D. Other classes: **Employee types, Facilities, Games, Customers, Tickets, and Rental info** as per your design in project 3

Input and code for reading input will be provided (same as project 4). Example code for writing output will also be provided (same as given for project 4). We may provide a different larger input for final usage or for testing. Use Proj5Constants.java (same as Proj4Constants.java) as well.

What you need to accomplish in this project (and will be graded on). You will use project 4 to process all the commands:

A. You will implement a class called GameComparator that **implements** the `Comparator<Game>` interface. This class will implement **one public method** int `compare(Game game1, Game game2)` method that takes 2 Game objects and returns a positive, zero, or negative integer, respectively, if the date of of game1 is later, same, or earlier than the date of game2. You then use the `Collections.sort(gameArrayList, new GameComparator());` statement to sort the gameArrayList (containing game objects) on their date of game for the new input command (# 19 in the data input file) with no parameters

and will display all games in descending order of their date. gameArrayList is an ArrayList of type Game (i.e., gameArrayList<Game>). The compare method you will write may use the existing methods of the Date class to compare dates.
The Comparator interface is defined as follows in Java

```
interface Comparator<T>{
  public int compare(<T> obj1, <T> obj2);
}
```

B. You will store the rental information of all facilities after processing all commands by using the `HashMap<K, V>` data structure provided by the Java Collections framework. This is done only once, the first time you encounter command 20. You will store the rental information of each facility in a Hashmap as value using the facility name (String, all lowercase with spaces trimmed) as the key. If a facility has been rented multiple times (over the duration of the data set) then all of the rental information for that facility is stored with that key. Since the same facility can be rented many a times over a period, the value component of the HashMap will be an ArrayList of this information (i.e., type RentalInfo). An input command (#20 with one parameter being the facility name as string or *) is added to the input file. When encountered, you will display rental details of that facility in ascending Date order in which that facility was rented. If * is given, do the above for all facilities in the HashMap. You can get a list of entries in the HashMap<K, V> by using the method `public Collection<V> values()` which can be iterated to get all the values.

   For sorting the `ArrayList<RentalInfo>` which is the `V` of the `HashMap`, use the comparable object approach by implementing a `int compareTo(RentalInfo ri)` instance method in the `RentalInfo` class. This will return a positive, zero, or a negative integer depending upon whether the Date of `this` is greater than, equal to, or less than the Date of ri (see lecture notes).

C. You can populate the HashMap when this command is encountered for the **first time**. You will iterate through all RentalInfo data structure to populate the HashMap whose value is an ArrayList of RentalInfo. At the end, you will sort (low to high) on the Date value of the rental. This should give a sorted date order of that facility as rented. This will help the MavStadium management to Identify preferences of a facility based on how often it is rented in a year.

Documentation, Coding style, and Grading Scheme:

A. Java provides mechanisms for comments and input for Javadoc. Please make sure your code contains both. Also, submit the files generated by the Javadoc for your program. Look up the convention for specifying parameters (@param), @override.

and user information for javadoc. Every method should have its meaningful description for use by someone else. Use the relevant Javadoc tags specified in - http://en.wikipedia.org/wiki/Javadoc or http://java.sun.com/j2se/javadoc/writingdoccomments/

B. In addition, you will report a few (3 to 5) important logical (not syntactic) errors that you encountered while doing this project. Furthermore, you will briefly provide an explanation on how you debugged and fixed each of these errors.

Coding Style:

Be sure to observe standard Java naming conventions and style. These will be observed across all projects for this course, hence it is necessary that you understand and follow them correctly. They include:

a. Class names begin with an upper-case letter, as do any subsequent words in the class name.
b. Method names begin with a lower-case letter, and any subsequent words in the method name begin with an upper-case letter.
c. Class, instance and local variables begin with a lower-case letter, and any subsequent words in the name of that variable begin with an upper-case letter.
d. Each set of opening/closing curly braces must be in the same column. All user prompts must be clear and understandable
e. Give meaningful names for classes, methods, and variables even if they seem to be long. The point is that the names should be easy to understand for a person looking at your code
f. Your program is properly indented to make it understandable. Proper matching of if … then … else and other control structures is important and should be easily understandable
g. Do not put multiple statements in a single line
h. All *static final* variables will be upper case identifiers
i. There will **no hardwired constants** used in the programs. We have provided a constants file to be used for the project (Proj5Constants.java (same as Proj3Constants.java))
j. Please use a main method in **each** class for unit testing that class

In addition, ensure that your code is properly documented (apart from javadoc commands) in terms of comments and other forms of documentation wherever necessary.