

```

1. import java.time.LocalDateTime;
2. import java.util.ArrayList;
3. public class Booking {
4.
5.     enum StatusValue{
6.         SCHEDULED,
7.         COMPLETED
8.     }
9.     private BookableRoom bookableRoom;
10.         private AssistantOnShift assistantOnShift;
11.         private int identifiatiionCode;
12.
13.         private String studentEmail;
14.         private StatusValue status = StatusValue.SCHEDULED;
15.         private static Booking[] bookings = new Booking[100];
16.         private static int numBookings = 0;
17.         /**
18.          * The constructor for the bookings of the tests.
19.          *
20.          * @param identifiatiionCode the code to identify the
21.          booking.
22.          * @param studentEmail the email of the student who
23.          will be tested.
24.          * @param bookableRoom the room that has been booked
25.          for the test.
26.          * @param assistantOnShift the assistant that will be
27.          doing the test.
28.          */
29.         public Booking(int identifiatiionCode, String
30.             studentEmail,
31.             BookableRoom bookableRoom, AssistantOnShift
32.             assistantOnShift){
33.             this.identifiatiionCode =
34.             checkIdentificationCode(identifiatiionCode);
35.             this.studentEmail = checkEmail(studentEmail);
36.             this.bookableRoom = checkBookableRoom(bookableRoom);
37.             this.assistantOnShift =
38.             checkAssistantOnShift(assistantOnShift);
39.             checkTimeSlot();
40.             assistantOnShift.setStatus(true);
41.             bookableRoom.setOccupancy(bookableRoom.getOccupancy()
42. +1);
43.             addBookings(this);
44.             iterateNumBookings();
45.         }
46.         /**
47.          *
48.          * The getter method for the room this test is booked
49.          in.
50.          *
51.          * @return the bookable room.
52.          */
53.         public BookableRoom getBookableRoom(){
54.             return bookableRoom;
55.         }

```

```

45.      /**
46.      * The getter method for the assistant that will be
    doing the test.
47.      *
48.      * @return the assistant on shift.
49.      */
50.      public AssistantOnShift getAssistantOnShift(){
51.      return assistantOnShift;
52.      }
53.      /**
54.      * The getter method for the identification code
55.      *
56.      * @return the code to identify the booking.
57.      */
58.      public int getIdentificationCode(){
59.      return identifiatiionCode;
60.      }
61.      /**
62.      * The getter method for the student's email.
63.      *
64.      * @return the email of the student.
65.      */
66.      public String getStudentEmail(){
67.      return studentEmail;
68.      }
69.      /**
70.      * The getter method for the status of the booking.
71.      *
72.      * @return the current status of the booking.
73.      */
74.      public StatusValue getStatus(){
75.      return status;
76.      }
77.      /**
78.      * The static getter method for the number of
    bookings.
79.      *
80.      * @return the number of bookings.
81.      */
82.      public static int getNumBookings(){
83.      return numBookings;
84.      }
85.      /**
86.      * The static getter method for the list of all
    bookings.
87.      *
88.      * @return the list of all bookings.
89.      */
90.      public static Booking[] getBookings(){
91.      return bookings;
92.      }
93.      /**
94.      * The setter method for the identification code
95.      *

```

```

96.      * @param identificationCode a new identification code
      for the booking to be set
97.      to.
98.      */
99.      public void setIdentificationCode(int
      identificationCode){
100.         this.identificationCode =
      checkIdentificationCode(identificationCode);
101.     }
102.     /**
103.     * The setter method for the student's email.
104.     *
105.     * @param studentEmail a new student's email for this
      booking.
106.     */
107.     public void setStudentEmail(String studentEmail){
108.         this.studentEmail = checkEmail(studentEmail);
109.     }
110.     /**
111.     * The setter method for the current status of the
      booking
112.     *
113.     * @param complete true - the booking is complete,
      false - the booking is
114.     scheduled.
115.     */
116.     public void setStatus(boolean complete){
117.         if (complete){//The status can only go from scheduled
      to complete not the
118.         other way round.
119.         this.status = StatusValue.COMPLETED;
120.         }
121.     }
122.     /**
123.     * The static method to incement the number of
      bookings.
124.     */
125.     private static void iterateNumBookings(){
126.         numBookings += 1;
127.     }
128.     /**
129.     * The static method to add a new booking to the list
      of bookings
130.     *
131.     * @param booking a new booking to be added to the
      list.
132.     */
133.     private static void addBookings(Booking booking){
134.         bookings[numBookings] = booking;
135.     }
136.     /**
137.     * The static method to remove a booking from the list
      of bookings
138.     *

```

```

139.      * @param booking a booking to be removed from the
      list.
140.      */
141.      public static void removeBookings(Booking booking){
142.          if (booking.status != StatusValue.SCHEDULED){
143.              throw new IllegalArgumentException("To remove the
              booking it cannot
144.              already been completed.");
145.          }
146.          boolean found = false;
147.          int index = -1;
148.          // linear search to find booking in the array.
149.          while (!found){
150.              index += 1;
151.              if (bookings[index] == booking){
152.                  found = true;
153.              }else if (index >= numBookings){
154.                  throw new IllegalArgumentException("The booking was
                  not found.");
155.              }
156.          }
157.          // Shifting the last elements.
158.          for (int i=index;i<=numBookings-1;i++){
159.              bookings[i] = bookings[i+1];
160.          }
161.          bookings[numBookings] = null;
162.          numBookings -= 1;
163.          }
164.      /**
165.       * The email checker private method.
166.       * Checks if the email of an student ends with
167.       * "@uok.ac.uk" and is unique.
168.       * @param email the email of a specific student.
169.       * @return the email of a specific student if it is
      valid.
170.      */
171.      private String checkEmail(String email){
172.          if (!email.endsWith("@uok.ac.uk")){
173.              throw new IllegalArgumentException("The email string
              should always end
174.              with @uok.ac.uk.");
175.          }
176.          for (int i=0;i<numBookings;i++){
177.              if (bookings[i].getStudentEmail().equals(email)){
178.                  throw new IllegalArgumentException("The email should
                  be
179.                  unique.");
180.              }
181.          }
182.          return email;
183.      }
184.      /**
185.       * The method to check the identification code is

```

```

    valid
186.     *
187.     * @param identifiatiionCode an identification code
    being checked.
188.     * @return the identfication code if it valid.
189.     */
190.     private int checkIdentificationCode(int
    identifiatiionCode){
191.         for (int i=0;i<numBookings;i++){
192.             if (bookings[i].getIdentificationCode() ==
    identifiatiionCode){
193.                 throw new IllegalArgumentException("The
    identification code should
194.                 be unique.");
195.             }
196.         }
197.         return identifiatiionCode;
198.     }
199.     /**
200.     * The method to check the bookable room is valid.
201.     *
202.     * @param bookableRoom a bookable room being checked.
203.     * @return the bookable room if it is valid.
204.     */
205.     private BookableRoom checkBookableRoom(BookableRoom
    bookableRoom){
206.         if
    (bookableRoom.getStatus().equals(BookableRoom.StatusValue.FU
    LL)){
207.             throw new IllegalArgumentException("The bookable room
    is full so not
208.             booking can be made.");
209.         }
210.         return bookableRoom;
211.     }
212.     /**
213.     * The method is check if an assistant on shift is
    valid.
214.     *
215.     * @param assistantOnShift an assistant on shift being
    check
216.     * @return the assistant on shift if it is valid.
217.     */
218.     private AssistantOnShift
    checkAssistantOnShift(AssistantOnShift
219.        assistantOnShift){
220.         if
221.         (assistantOnShift.getStatus().equals(AssistantOnShift.
    StatusValue.BUSY)){
222.             throw new IllegalArgumentException("The assistant on
    shift cannot be
223.             busy.");
224.         }
225.         return assistantOnShift;

```

```

226.     }
227.     /**
228.      * The method to check if the time slot is valid.
229.      */
230.     private void checkTimeSlot(){
231.         if (!
            assistantOnShift.getTimeSlot().isEqual(bookableRoom.getTimeS
            lot())){
232.             throw new IllegalArgumentException("The timeslots do
                not match with the
233.             assistant on shift and the bookable room.");
234.         }
235.     }
236.     /**
237.      * The static method to remove duplicates from a list.
238.      *
239.      * @param <e> the elements of that list.
240.      * @param list the list that contains duplicates.
241.      * @return the list that has no ducplices.
242.      */
243.     private static <e>ArrayList<e>
        removeDuplicate(ArrayList<e> list) {
244.         ArrayList<e> correctList = new ArrayList<e>(); //
            Uses array list since it
245.            is easier.
246.         for (e element : list) {
247.             if (!correctList.contains(element)) {
248.                 correctList.add(element);
249.             }
250.         }
251.         return correctList;
252.     }
253.     /**
254.      * The method to return the list of current valid time
        slots for new bookings.
255.      *
256.      * @return the list of valid time slots.
257.      */
258.     public static LocalDateTime[] validTimeSlots(){
259.         LocalDateTime[] validDateTime = new
260.         LocalDateTime[AssistantOnShift.getnumAssistantsOnShift
            s()];
261.         for (int
            i=0;i<AssistantOnShift.getnumAssistantsOnShifts();i++){
262.             for (int j=0;j<BookableRoom.getNumBookableRooms();j+
                +){
263.                 if
264.                 (AssistantOnShift.getAssistantOnShifts()
                    [i].getTimeSlot().equals(BookableRoom.getBo
265.                    okableRooms()[j].getTimeSlot())
266.                    &&
267.                    AssistantOnShift.getAssistantOnShifts()
                    [i].getStatus().equals(AssistantOnShift.Stat
268.                    usValue.FREE)

```

```

269.         &&
270.         !BookableRoom.getBookableRooms()
271.         [j].getStatus().equals(BookableRoom.StatusValue.FUL
272.         L)){
273.             validDateTime[i] =
274.             AssistantOnShift.getAssistantOnShifts()
275.             [i].getTimeSlot();
276.             break;
277.         }
278.         return validDateTime;
279.     }
280.     /**
281.     * The method to return the string of all valid time
282.     slots.
283.     * @return the string of all valid time slots in the
284.     correct format.
285.     */
286.     public static String toStringTimeSlots(){
287.         String allTimeSlots = "List of available time-
288.         slots-\n";
289.         ArrayList<LocalDateTime> validDateTime = new
290.         ArrayList<LocalDateTime>();
291.         for (int
292.         i=0;i<AssistantOnShift.getnumAssistantsOnShifts();i++){
293.             for (int j=0;j<BookableRoom.getNumBookableRooms();j+
294.             +){
295.                 if
296.                 (AssistantOnShift.getAssistantOnShifts()
297.                 [i].getTimeSlot().equals(BookableRoom.getBo
298.                 okableRooms()[j].getTimeSlot())
299.                 &&
300.                 AssistantOnShift.getAssistantOnShifts()
301.                 [i].getStatus().equals(AssistantOnShift.Stat
302.                 usValue.FREE)
303.                 &&
304.                 !BookableRoom.getBookableRooms()
305.                 [j].getStatus().equals(BookableRoom.StatusValue.FUL
306.                 L)){
307.                     validDateTime.add(AssistantOnShift.getAssistantOnShift
308.                     s()[i].getTimeSlot());
309.                     break;
310.                 }
311.             }
312.         }
313.         validDateTime = removeDuplicate(validDateTime);
314.         for (int i=0;i<validDateTime.size();i++){
315.             allTimeSlots = allTimeSlots.concat((i+11)+"
316.             "+BookableRoom.formatDate(validDateTime.get(i))+"\n");
317.         }
318.         return allTimeSlots;

```

```

311.     }
312.     /**
313.      * This is the overloaded method to return the the
314.      * string of all bookings.
315.      * @return a string of all bookings.
316.      */
317.     public static String toStringAll(){
318.         String allBookings = "Bookings-\n";
319.         for (int i=0;i<numBookings;i++){
320.             allBookings = allBookings.concat((i+11)+".
321.             "+bookings[i].toString()+"\n");
322.         }
323.         return allBookings;
324.     }
325.     /**
326.      * This is the overloaded method to return the the
327.      * string of all bookings
328.      * @param status this is the status to filter the
329.      * bookings by.
330.      * @return a string of bookings that are valid for the
331.      * status.
332.      */
333.     public static String toStringAll(StatusValue status){
334.         String allbookings = "Bookings-\n";
335.         int index = 0;
336.         for (int i=0;i<numBookings;i++){
337.             if (bookings[i].status == status){
338.                 allbookings = allbookings.concat((index+11)+".
339.                 "+bookings[i].toString()+"\n");
340.                 index++;
341.             }
342.         }
343.         return allbookings;
344.     }
345.     /**
346.      * This is the method to return a string of a booking.
347.      * @return a string of a booking.
348.      */
349.     public String toString(){
350.         return "|
351.         "+assistantOnShift.formatDate(assistantOnShift.getTimeSlot())
352.         )+
353.         " | "+status+" |
354.         "+assistantOnShift.getAssistant().getEmail()+" |
355.         "+bookableRoom.getRoom().getCode()+" |
356.         "+studentEmail+" |";
357.     }
358. }

```