UNIVERSITY OF BOLTON
BSC COMPUTING
COURSEWORK SUBMISSION FORM

Student/Centre to complete:

SURNAME/FAMILY NAME: Ellahi Begum FORENAMES: Sami Ullah

BOLTON STUDENT ID: 2116149 EMAIL: su4crt@bolton.ac.uk

DATE OF SUBMISSION: 03/11/2023

MODULE NO./TITLE:…SWE5202 Data Structures and Algorithms

TUTOR'S NAME: Abdul Razak……………………..... ……………………

COURSEWORK TITLE: Portfolio item 1: Cloning, Sorting, Searching, shallow and deep copying of objects.
Please state if this is your FIRST submission OR REFERRED/DEFERRED submission
OR a REPEAT submission?

FIRST…………………………………………………………………………………………

Declaration
I hereby declare that this work is my own work. I understand that if I am suspected of plagiarism or another form of cheating, my work be referred to Academic Registrar and/or the Board of Examiners, which may result in me being expelled from the programme. I understand once I submit this work, it will automatically belong to the University of Bolton.

Academic staff to complete:
Feedback: ……………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………
Date Issued: W/C 09 October 2023 Hand-In Date: **(27 October 2023 @ 16:00)**

Other Relevant Date e.g. Demonstration **In class demonstration on or before W/C 23 October 2023**

Received: On Time ☐ Late ☐ (within 5 days of published deadline date)
Mark awarded: ………..% Do not apply mark penalty unless the work was submitted late.

Assessors Name: …A. Razak………………… Signature:…Sami………………………

Date:…………………………
Degree Conversions A: 70-100% B: 60-69% C: 50-59% D: 40-49% F: 0-39%
HND Conversions Pass: 40-49% Merit: 50-66% Distinction: 67-100%
`

Late submission

For late submission, see Assessment Regulations for Undergraduate Programmes:

https://www.bolton.ac.uk/assets/Assessment-Regulations-for-Undergraduate-Programmes-2023-24-V10-v2.pdf

Creative Technologies	
Course / Programme:	**BEng (Hons) in Software Engineering**
Module name and code:	**Data Structures and Algorithms SWE5202**
Tutor:	**Abdul Razak**
Assessment Number:	**One**
Assessment Title:	**Cloning, Sorting, Searching, shallow and deep copying of objects.**
Weighting	**25%**
Issue Date:	**W/C 09 October 2023**
Submission Deadline:	**27 October 2023 @16.00.**

Learning Outcomes:

LO1: Implement sorting and searching algorithms for linear data structures.

LO2: Implement shallow and deep copying and use the Comparable and Comparator

Assessment:

Using cloning (both shallow and deep copying), sorting and searching algorithms on linear data structures.

HE5 – Assessment is set appropriate to level HE5.

Grading
A percentage mark will be provided as feedback. Grading is as follows:

A:	70-100%
B:	60-69%
C:	50-59%
D:	40-49%
F:	below 40%

Marks below 40% will be classed as fail.

Contents

PORTFOLIO ITEM 1

Create a new Java project called **A1_Donald_Duck** replacing ***Donald_Duck*** with your name. Use the underscore (_) to replace any space characters. In the project create the following 4 classes called Date, Time MeetingRoom and Appointment which can be used in an electronic organiser type application.

Class name: Date (Attributes)	
private int day	day of the month (1-31)
private int month	month of the year (1-12)
private int year	year e.g., 2008

Class name: Time (Attributes)	
private int hour	hour in the day (0-23)
private int minute	minute within the hour (0-59)

Class name: MeetingRoom (Attributes)	
private String room	room in String format e.g., B1-05 or C2-08 or T4-34

Class name: Appointment (Attributes)	
private String name	who to meet
private String purpose	purpose of appointment
private Date date	date of appointment
private Time time	time of appointment
private MeetingRoom room	where to meet

Program development stage 1

Purpose and Functionality:

In Stage 1 of our program development, we aimed to establish the foundational structure of our application by creating four essential classes: **Appointment**, **Date**, **Time**, and **MeetingRoom**. These classes lay the groundwork for managing and organizing appointments within our scheduling system.

Key Objectives:

1. **Default and Parameter Constructors:** To ensure versatility and flexibility, we implemented default constructors that initialize class attributes with sensible default values. These constructors provide a straightforward way to create instances of the classes. Additionally, we created parameterized constructors that allow users to set attribute values upon instantiation, facilitating customization.

2. **Copy Constructor (Shallow Copying):** For each class, we implemented a copy constructor that performs shallow copying. Shallow copying is useful for creating new instances while sharing references to existing objects, optimizing memory usage when dealing with potentially large data structures like dates, times, and room names.

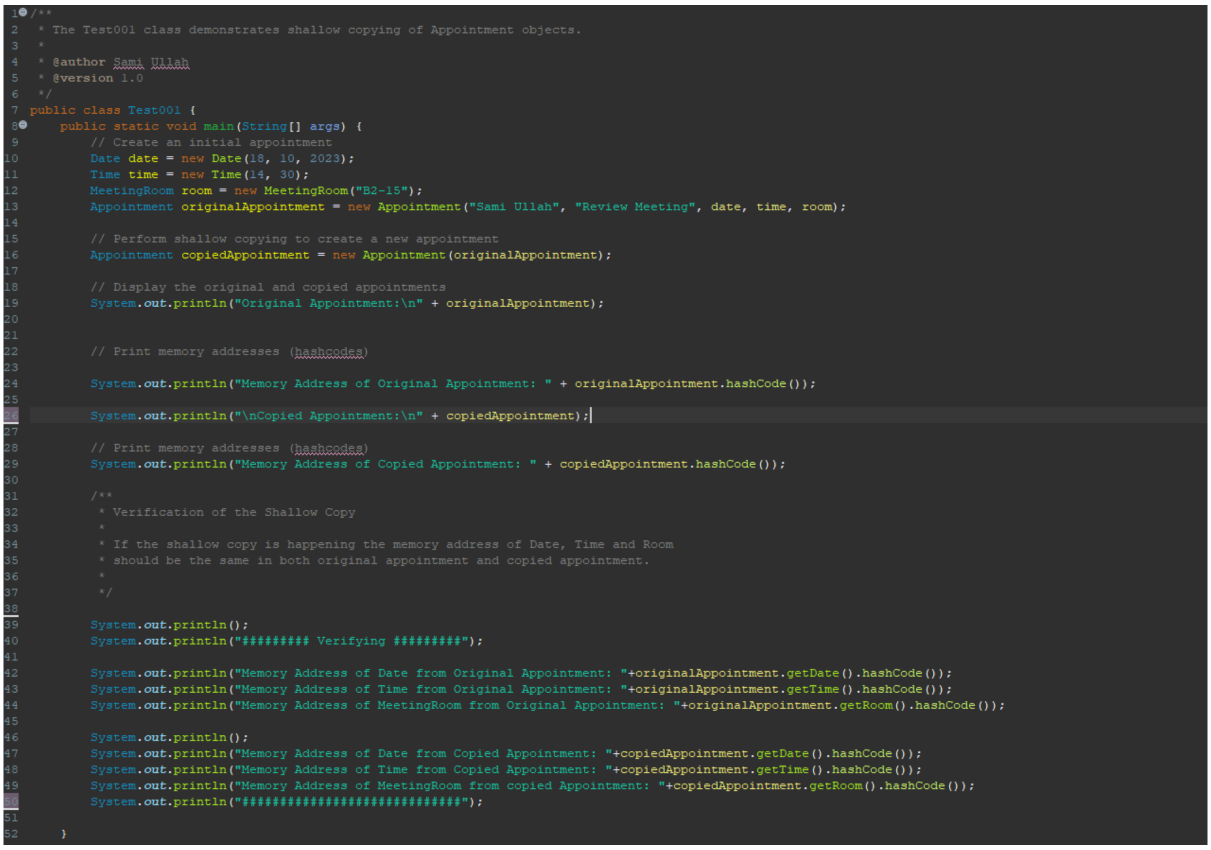
3. **Setter and Getter Methods:** We provided setter methods for each attribute, allowing users to modify object attributes as needed. Getter methods enable retrieving the attribute values.

4. **toString Method:** To facilitate the presentation and printing of object information, we implemented a **toString** method in each class, returning a suitably formatted string representation of attribute values.

Demonstration and Test Class:

To validate the functionality of these classes and to showcase shallow copying in action, we created a test class, Test001. This class serves as a practical example of how to work with these classes, demonstrating how to create, modify, and copy Appointment objects.

Test001



```java
/**
 * The Test001 class demonstrates shallow copying of Appointment objects.
 *
 * @author Sami Ullah
 * @version 1.0
 */
public class Test001 {
    public static void main(String[] args) {
        // Create an initial appointment
        Date date = new Date(18, 10, 2023);
        Time time = new Time(14, 30);
        MeetingRoom room = new MeetingRoom("B2-15");
        Appointment originalAppointment = new Appointment("Sami Ullah", "Review Meeting", date, time, room);

        // Perform shallow copying to create a new appointment
        Appointment copiedAppointment = new Appointment(originalAppointment);

        // Display the original and copied appointments
        System.out.println("Original Appointment:\n" + originalAppointment);


        // Print memory addresses (hashcodes)

        System.out.println("Memory Address of Original Appointment: " + originalAppointment.hashCode());

        System.out.println("\nCopied Appointment:\n" + copiedAppointment);

        // Print memory addresses (hashcodes)
        System.out.println("Memory Address of Copied Appointment: " + copiedAppointment.hashCode());

        /**
         * Verification of the Shallow Copy
         *
         * If the shallow copy is happening the memory address of Date, Time and Room
         * should be the same in both original appointment and copied appointment.
         *
         */

        System.out.println();
        System.out.println("######### Verifying #########");

        System.out.println("Memory Address of Date from Original Appointment: "+originalAppointment.getDate().hashCode());
        System.out.println("Memory Address of Time from Original Appointment: "+originalAppointment.getTime().hashCode());
        System.out.println("Memory Address of MeetingRoom from Original Appointment: "+originalAppointment.getRoom().hashCode());

        System.out.println();
        System.out.println("Memory Address of Date from Copied Appointment: "+copiedAppointment.getDate().hashCode());
        System.out.println("Memory Address of Time from Copied Appointment: "+copiedAppointment.getTime().hashCode());
        System.out.println("Memory Address of MeetingRoom from copied Appointment: "+copiedAppointment.getRoom().hashCode());
        System.out.println("#############################");

    }
```

In the Test001, to verify the shallow copy we have printed the memory address of the Date, Time and MeetingRoom of both original and copied appointments.

We can observe that all the memory addresses are the same. That means the shallow copy has been made successfully. In case they were different the shallow copy would have errors.

Appointment Class

```
import java.util.Comparator;
/**
* The Appointment class represents an appointment with a name, purpose, date,
time, and meeting room.
*
* @author Sami Ullah
* @version 1.0
*/
public class Appointment implements Cloneable, Comparable<Appointment>,
Comparator<Appointment>{
private String name; // who to meet
private String purpose; // purpose of appointment
private Date date; // date of appointment
private Time time; // time of appointment
private MeetingRoom room; // where to meet
/**
* Default constructor that initializes the appointment with default values.
*/
public Appointment() {
this.name = "John Doe";
this.purpose = "Meeting";
this.date = new Date();
this.time = new Time();
this.room = new MeetingRoom();
}
/**
* Parameter constructor that sets the attributes based on the provided
values.
*
* @param name the name of the person to meet
* @param purpose the purpose of the appointment
* @param date the date of the appointment
* @param time the time of the appointment
* @param room the meeting room for the appointment
*/
public Appointment(String name, String purpose, Date date, Time time,
MeetingRoom room) {
this.name = name;
this.purpose = purpose;
this.date = date;
this.time = time;
this.room = room;
}
/**
* Copy constructor that performs shallow copying of the appointment.
*
* @param other the appointment to copy
*/
public Appointment(Appointment other) {
this.name = other.name;
this.purpose = other.purpose;
this.date = (other.date);
this.time = (other.time);
this.room = (other.room);
}
/**
* Gets the name of the person to meet.
*
* @return the name of the person
```

```
*/
public String getName() {
return name;
}
/**
* Sets the name of the person to meet.
*
* @param name the name of the person to set
*/
public void setName(String name) {
this.name = name;
}
/**
* Gets the purpose of the appointment.
*
* @return the purpose of the appointment
*/
public String getPurpose() {
return purpose;
}
/**
* Sets the purpose of the appointment.
*
* @param purpose the purpose of the appointment to set
*/
public void setPurpose(String purpose) {
this.purpose = purpose;
}
/**
* Gets the date of the appointment.
*
* @return the date of the appointment
*/
public Date getDate() {
return date;
}
/**
* Sets the date of the appointment.
*
* @param date the date of the appointment to set
*/
public void setDate(Date date) {
this.date = date;
}
/**
* Gets the time of the appointment.
*
* @return the time of the appointment
*/
public Time getTime() {
return time;
}
/**
* Sets the time of the appointment.
*
* @param time the time of the appointment to set
*/
public void setTime(Time time) {
this.time = time;
}
/**
```

```
* Gets the meeting room for the appointment.
*
* @return the meeting room for the appointment
*/
public MeetingRoom getRoom() {
return room;
}
/**
* Sets the meeting room for the appointment.
*
* @param room the meeting room for the appointment to set
*/
public void setRoom(MeetingRoom room) {
this.room = room;
}
/**
* Creates a deep copy of the appointment object.
*
* @return a deep copy of the appointment
*/
@Override
public Object clone() {
try {
Appointment copy = (Appointment) super.clone();
copy.date = (Date) date.clone();
copy.time = (Time) time.clone();
copy.room = (MeetingRoom) room.clone();
return copy;
} catch (CloneNotSupportedException e) {
throw new InternalError(e);
}
}
/**
* Compares this appointment to another appointment for natural ordering based
on date and time.
*
* @param other the other appointment to compare to
* @return a negative integer, zero, or a positive integer as this appointment
is earlier, the same, or later than the other
*/
// @Override
// public int compareTo(Appointment other) {
// long thisDateTime = this.dateToLong() * 10000L + this.timeToLong();
// long otherDateTime = other.dateToLong() * 10000L + other.timeToLong();
// return Long.compare(thisDateTime, otherDateTime);
// }
@Override
public int compareTo(Appointment other) {
int dateComparison = this.date.compareTo(other.date);
if (dateComparison != 0) {
return dateComparison;
} else {
int timeComparison = this.time.compareTo(other.time);
if (timeComparison != 0) {
return timeComparison;
} else {
return this.room.compareTo(other.room); //Added as a backup option
}
}
}
// Helper methods to convert date and time to long values
// private long dateToLong() {
//  return  this.date.getYear()  *  10000L  +  this.date.getMonth()  *  100L  +
this.date.getDay();
// }
//
// private long timeToLong() {
// return this.time.getHour() * 100L + this.time.getMinute();
// }
@Override
public int compare(Appointment appointment1, Appointment appointment2) {
String room1Floor = appointment1.getRoom().getRoom().substring(1, 2);
String room2Floor = appointment2.getRoom().getRoom().substring(1, 2);
int floor1 = Integer.parseInt(room1Floor);
int floor2 = Integer.parseInt(room2Floor);
return Integer.compare(floor1, floor2);
}
/**
* Returns a string representation of the appointment's attributes.
*
* @return a formatted string with appointment details
*/
@Override
public String toString() {
return "Appointment Details:\n" +
"Name: " + name + "\n" +
"Purpose: " + purpose + "\n" +
"Date: " + date + "\n" +
"Time: " + time + "\n" +
"Room: " + room;
}
}
```

Date Class

```
/**
* The Date class represents a date with day, month, and year.
*
* @author Sami Ullah
* @version 1.0
*/
public class Date implements Cloneable{
private int day; // day of the month (1-31)
private int month; // month of the year (1-12)
private int year; // year (e.g., 2008)
/**
* Default constructor that initializes the date to January 1, 2000.
*/
public Date() {
this.day = 1;
this.month = 1;
this.year = 2000;
}
/**
* Parameter constructor that sets the day, month, and year based on the
provided values.
*
* @param day the day of the month (1-31)
* @param month the month of the year (1-12)
* @param year the year value (e.g., 2008)
*/
```

```java
public Date(int day, int month, int year) {
this.day = day;
this.month = month;
this.year = year;
}
/**
* Copy constructor that performs shallow copying of the date.
*
* @param other the date to copy
*/
public Date(Date other) {
this.day = other.day;
this.month = other.month;
this.year = other.year;
}
/**
* Gets the day of the month.
*
* @return the day of the month (1-31)
*/
public int getDay() {
return day;
}
/**
* Sets the day of the month.
*
* @param day the day of the month to set (1-31)
*/
public void setDay(int day) {
this.day = day;
}
/**
* Gets the month of the year.
*
* @return the month of the year (1-12)
*/
public int getMonth() {
return month;
}
/**
* Sets the month of the year.
*
* @param month the month of the year to set (1-12)
*/
public void setMonth(int month) {
this.month = month;
}
/**
* Gets the year.
*
* @return the year value (e.g., 2008)
*/
public int getYear() {
return year;
}
/**
* Sets the year.
*
* @param year the year value to set (e.g., 2008)
*/
public void setYear(int year) {
```

```java
this.year = year;
}
/**
* Creates a deep copy of the date object.
*
* @return a deep copy of the date
*/
@Override
public Object clone() {
try {
return super.clone();
} catch (CloneNotSupportedException e) {
throw new InternalError(e);
}
}
public int compareTo(Date other) {
long thisValue = year * 10000L + month * 100L + day;
long otherValue = other.year * 10000L + other.month * 100L + other.day;
return Long.compare(thisValue, otherValue);
}
/**
* Returns a formatted string representation of the date.
*
* @return a string in the "dd-MM-yyyy" format
*/
@Override
public String toString() {
return String.format("%02d-%02d-%04d", day, month, year);
}
}
```

Time Class

```java
/**
* The Time class represents a time with hours and minutes.
*
* @author Sami Ullah
* @version 1.0
*/
public class Time implements Cloneable{
private int hour; // hour in the day (0-23)
private int minute; // minute within the hour (0-59)
/**
* Default constructor that initializes the time to 00:00.
*/
public Time() {
this.hour = 0;
this.minute = 0;
}
/**
* Parameter constructor that sets the hour and minute based on the provided
values.
*
* @param hour the hour value (0-23)
* @param minute the minute value (0-59)
*/
public Time(int hour, int minute) {
this.hour = hour;
this.minute = minute;
}
/**
* Copy constructor that performs shallow copying of the time.
*
* @param other the time to copy
*/
public Time(Time other) {
this.hour = other.hour;
this.minute = other.minute;
}
/**
* Gets the hour value.
*
* @return the hour value (0-23)
*/
public int getHour() {
return hour;
}
/**
* Sets the hour value.
*
* @param hour the hour value to set (0-23)
*/
public void setHour(int hour) {
this.hour = hour;
}
/**
* Gets the minute value.
*
* @return the minute value (0-59)
*/
public int getMinute() {
return minute;
}
/**
* Sets the minute value.
*
* @param minute the minute value to set (0-59)
*/
public void setMinute(int minute) {
this.minute = minute;
}
/**
* Creates a deep copy of the time object.
*
* @return a deep copy of the time
*/
@Override
public Object clone() {
try {
return super.clone();
} catch (CloneNotSupportedException e) {
throw new InternalError(e);
}
}
public int compareTo(Time other) {
// Construct a Long value representing the time (HHMM)
long thisValue = hour * 100L + minute;
long otherValue = other.hour * 100L + other.minute;
return Long.compare(thisValue, otherValue);
}
```

```
/**
* Returns a formatted string representation of the time.
*
* @return a string in the "HH:mm" format
*/
@Override
public String toString() {
return String.format("%02d:%02d", hour, minute);
}
}
```

MeetingRoom Class

```
/**
* The MeetingRoom class represents a meeting room with a room name.
*
* @author Sami Ullah
* @version 1.0
*/
public class MeetingRoom implements Cloneable{
private String room; // room in String format (e.g., B1-05, C2-08, T4-34)
/**
* Default constructor that initializes the meeting room with a default
name.
*/
public MeetingRoom() {
this.room = "A1-01";
}
/**
* Parameter constructor that sets the room name based on the provided
value.
*
* @param room the room name
*/
public MeetingRoom(String room) {
this.room = room;
}
/**
* Copy constructor that performs shallow copying of the meeting room.
*
* @param other the meeting room to copy
*/
public MeetingRoom(MeetingRoom other) {
this.room = other.room;
}
/**
* Gets the room name.
*
* @return the room name
*/
public String getRoom() {
return room;
}
/**
* Sets the room name.
```

```java
*
* @param room the room name to set
*/
public void setRoom(String room) {
this.room = room;
}
/**
* Creates a deep copy of the meeting room object.
*
* @return a deep copy of the meeting room
*/
@Override
public Object clone() {
try {
return super.clone();
} catch (CloneNotSupportedException e) {
throw new InternalError(e);
}
}
public int compareTo(MeetingRoom other) {
return room.compareTo(other.room);
}
/**
* Returns the room name as a string.
*
* @return the room name
*/
@Override
public String toString() {
return room;
}
}
```

Program development stage 2

Purpose and Functionality:

Stage 2 of our program development focuses on the concept of deep copying for **Appointment** objects. The goal is to ensure that when creating a copy of an appointment, we create a completely new instance that is entirely independent of the original object. Deep copying guarantees that any changes made to the copied appointment do not affect the original, maintaining data integrity and isolation.

Key Objectives:

1. **Understanding Deep Copying:** Deep copying, in contrast to shallow copying, creates entirely independent objects. In our implementation, this means that copying an **Appointment** results in a new appointment object with the same attribute values, but without any shared references to underlying objects (e.g., **Date**, **Time**, or **MeetingRoom** instances).

2. **Test and Verification:** We designed a test class, **Test002**, to create multiple **Appointment** objects and then demonstrate deep copying. The primary objective is to validate that deep copying has been effectively implemented in our system.

Demonstration and Test Class:

The **Test002** class serves as a practical example of how deep copying works. In this stage, we created various **Appointment** objects with distinct attributes. Subsequently, we verified the deep copying mechanism by creating copies of these appointments and modifying the copied instances.

```java
/**
* The Test002 class demonstrates deep copying of Appointment objects.
*
* @author Sami Ullah
* @version 1.0
*
*/
public class Test002 {
public static void main(String[] args) {
// Create an initial appointment
Date date = new Date(29, 9, 2023);
Time time = new Time(14, 30);
MeetingRoom room = new MeetingRoom("B2-15");
Appointment originalAppointment = new Appointment("Sami Ullah", "Review
Meeting", date, time, room);
// Perform deep copying to create a new appointment
Appointment copiedAppointment = (Appointment) originalAppointment.clone();
// Modify the copied appointment
// copiedAppointment.setName(" Sam Naveed");
// copiedAppointment.getDate().setDay(29);
// copiedAppointment.getTime().setHour(15);
// copiedAppointment.getRoom().setRoom("C3-01");
// Display the original and copied appointments
System.out.println("Original Appointment:\n" + originalAppointment);
// Print memory addresses (hashcodes)
System.out.println("Memory Address of Original Appointment: " +
originalAppointment.hashCode());
System.out.println("\nCopied Appointment:\n" + copiedAppointment);
// Print memory addresses (hashcodes)
System.out.println("Memory Address of Copied Appointment: " +
copiedAppointment.hashCode());
/**
* Testing and verifying the deep copy has been archived.
*
*/
System.out.println();
System.out.println("########## Verifying Deepcopy ###########");
Date date2 = new Date(21, 11, 2023);
Time time2 = new Time(14, 30);
MeetingRoom room2 = new MeetingRoom("B1-08");
Appointment originalAppointment2 = new Appointment("John Henson", "Review
Meeting", date2, time2, room2);
// Perform deep copying to create a new appointment
Appointment copiedAppointment2 = (Appointment)
originalAppointment2.clone();
// Modify the copied appointment
copiedAppointment2.setName("Hafsa Khatoon");
copiedAppointment2.getTime().setHour(15);
copiedAppointment2.getRoom().setRoom("C2-13");
// Display the original and copied appointments
System.out.println("Original Appointment2:\n" + originalAppointment2);
System.out.println("\nCopied Appointment2 (with modifications):\n" +
copiedAppointment2);
System.out.println();
System.out.println("Memory Address of Date from Original Appointment:
"+originalAppointment2.getDate().hashCode());
System.out.println("Memory Address of Time from Original Appointment:
"+originalAppointment2.getTime().hashCode());
System.out.println("Memory Address of MeetingRoom from Original
Appointment: "+originalAppointment2.getRoom().hashCode());
System.out.println();
System.out.println("Memory Address of Date from Copied Appointment:
"+copiedAppointment2.getDate().hashCode());
System.out.println("Memory Address of Time from Copied Appointment:
"+copiedAppointment2.getTime().hashCode());
System.out.println("Memory Address of MeetingRoom from copied Appointment:
"+copiedAppointment2.getRoom().hashCode());
System.out.println();
System.out.println("###### Verification Has Been Completed ######");
}
}
```

Verification Process:

To ensure the accuracy of deep copying, we compared the memory addresses (hash codes) of the original **Appointment** objects and their corresponding copies. In deep copying, these memory addresses should be different, signifying that new, separate instances have been created.

Deep copying ensures that when a copy of an object is created, all referenced objects within the original object are also duplicated, so changes in the copied object do not affect the original object. To test and verify that deep copying has been achieved in the provided code, we can compare the original and copied objects before and after making modifications.

In the **Test002** class, we created an initial **Appointment** object and performed deep copying to create a new appointment. We then made modifications to the copied appointment and compared both the original and copied appointments. To test the deep copying, follow these steps:

1. Create an **Appointment** object named **originalAppointment** with initial values.
2. Perform deep copying to create a new **Appointment** object named **copiedAppointment**.
3. Modify the **copiedAppointment** by changing its attributes (name, date, time, and room).
4. Compare the original and copied appointments before and after modifications.

Here's an explanation of the test:

1. **Original Appointment**:
    - **originalAppointment2** contains the initial values: name "John Henson," date "21-11-2023," time "14:30," and room "B1-08."
2. **Deep Copying**:
    - The **Appointment** class's **clone()** method is implemented for deep copying.

- This method creates new instances of **Date2**, **Time2**, and **MeetingRoom2** to ensure that changes in the copied object won't affect the original.

3. **Modified Copied Appointment**:
    - We change the name to "Hafsa Khatoon", time's hour to "15," and room to "C2-13" in the **copiedAppointment2**.
4. **Comparison**:
    - We print both the original and copied appointments before and after modifications to see if changes in the copied appointment affect the original appointment.

The test demonstrates that deep copying has been achieved because the modifications made to the **copiedAppointment2** do not affect the **originalAppointment2**. The two objects remain independent after the modifications.

This verification aligns with the concept of deep copying, ensuring that referenced objects (e.g., **Date2**, **Time2**, **MeetingRoom2**) are duplicated, and changes in one object do not impact the other. This ensures data integrity and separation between the original and copied objects. Deep copying is a fundamental concept in object-oriented programming for maintaining data encapsulation and immutability.

Conclusion:

Stage 2 demonstrates the successful implementation of deep copying for **Appointment** objects. Our test cases confirm that the copied objects are entirely separate from the originals, maintaining data integrity and object isolation. This feature is fundamental in ensuring that modifications to one appointment do not affect others. The deep copying mechanism is a crucial step towards building a robust and reliable appointment management system.

Program development stage 3

Purpose and Functionality:

The focus of Stage 3 in our program development is to implement the **Comparable** interface in all four classes, namely **Appointment**, **Date**, **Time**, and **MeetingRoom**. The primary objective is to facilitate natural ordering of **Appointment** objects based on chronological order, which encompasses the date and time attributes.

Implementation of Natural Ordering:

Appointment Class: To enable natural ordering of **Appointment** objects, we implement the **Comparable** interface in the **Appointment** class. We introduce the **compareTo** method, which constructs a **Long** integer representing the date and time, and sorts the **Appointment** objects accordingly. The **compareTo** method is designed to compare two **Appointment** objects based on the chronological order, ensuring that the earliest appointment comes first.

Date and Time Classes: In the **Date** and **Time** classes, we also implement the **Comparable** interface to allow natural ordering of date and time attributes. These classes now have a **compareTo** method that compares their respective values. By adhering to natural ordering, we ensure that dates and times are sorted correctly.

MeetingRoom Class: Similar to other classes, the **MeetingRoom** class implements the **Comparable** interface. It defines a **compareTo** method that compares **MeetingRoom** objects based on their room names. Natural ordering ensures that meeting rooms are sorted alphabetically.

Demonstration and Test Class:

To validate the implementation, we created a test class, **Test003**, that creates multiple **Appointment** objects and stores them in both an **ArrayList** and an array. These objects are in the same original order as in the **ArrayList**. In addition, we assigned references to an unsorted **Appointment** object, **searchAppointment**, and its clone, **searchAppointmentClone**.

We used the for-each loop with the **ArrayList** to demonstrate the sorting of **Appointment** objects. The built-in sorting mechanism sorts the appointments in chronological order based on the date and time attributes.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
/**
* The Test003 class demonstrates natural ordering and stores appointments
in ArrayList and arrays.
*
* @author Sami Ullah
* @version 1.0
*/
public class Test003 {
public static void main(String[] args) {
// Create ArrayList of Appointment objects
ArrayList<Appointment> appointmentList = new ArrayList<Appointment>();
// Create appointments (same as before)
// Add appointments to the list
appointmentList.add(new Appointment("Josh Anderson", "Presentation 1", new
Date(15, 10, 2023), new Time(8, 30), new MeetingRoom("B2-03")));
appointmentList.add(new Appointment("John Smith", "Demo 1", new Date(19,
10, 2023), new Time(10, 15), new MeetingRoom("A1-01")));
appointmentList.add(new Appointment("Usman Khan", "Demonstration 3", new
Date(3, 10, 2023), new Time(13, 45), new MeetingRoom("C1-10")));
appointmentList.add(new Appointment("Nick Ross", "Num 1", new Date(23, 10,
2023), new Time(12, 45), new MeetingRoom("D3-05")));
appointmentList.add(new Appointment("James Red", "Meeting 1", new Date(20,
10, 2023), new Time(10, 45), new MeetingRoom("E3-08")));
appointmentList.add(new Appointment("Romeo Julliet", "Demo 2", new Date(1,
9, 2023), new Time(13, 15), new MeetingRoom("D1-05")));
appointmentList.add(new Appointment("Khan Ghazi", "Demo 3", new Date(20, 8,
2023), new Time(15, 10), new MeetingRoom("C2-10")));
appointmentList.add(new Appointment("Marc Clarke", "Meeting 2", new
Date(20, 11, 2023), new Time(11, 55), new MeetingRoom("E3-13")));
appointmentList.add(new Appointment("Nick Jonas", "Discusion", new Date(3,
10, 2023), new Time(19, 30), new MeetingRoom("A1-08")));
appointmentList.add(new Appointment("Naveed Sabir", "Meeting 3", new
Date(7, 10, 2023), new Time(9, 45), new MeetingRoom("B1-08")));
// Create an array of Appointment objects (same objects, in the same
original order as ArrayList)
Appointment[] appointmentArray = appointmentList.toArray(new
Appointment[appointmentList.size()]);
// Display appointments in chronological order from ArrayList (using for-
each loop) before sorting
System.out.println("####### LIST OF ALL APPOINTMENTS ##########");
System.out.println("Appointments (ArrayList) Before Sorting:");
int count=0;
for (Appointment appointment : appointmentList) {
System.out.println("Appointment Number: " +count);
System.out.println(appointment);
count+=1;
System.out.println();
}
/**
* Sort the ArrayList for natural ordering (based on the compareTo method)
```

```java
*/
Collections.sort(appointmentList);
/**
* Display appointments in sorted order (ArrayList)
*/
System.out.println("####### LIST OF ALL APPOINTMENTS AFTER SORTING
#######");
int count2=0;
System.out.println("\nAppointments (ArrayList) Sorted chonologically:");
for (Appointment appointment : appointmentList) {
System.out.println("Appointment Number: " +count2);
System.out.println(appointment);
count2+=1;
System.out.println();
}
System.out.println("####### LIST OF ALL APPOINTMENTS #######");
// Display appointments in chronological order from the array
int count3=0;
System.out.println("\nAppointments (Array) in Chronological Order:");
for (Appointment appointment : appointmentArray) {
System.out.println("Appointment Number: " +count3);
System.out.println(appointment);
count3+=1;
System.out.println();
}
// Sort the array for natural ordering (based on the compareTo method)
Arrays.sort(appointmentArray);
System.out.println("####### LIST OF ALL APPOINTMENTS AFTER SORTING
#######");
int count4=0;
// Display appointments in chronological order from the array after sorting
System.out.println("\nAppointments (Array) in Chronological Order (after
sorting):");
for (Appointment appointment : appointmentArray) {
System.out.println("Appointment Number: " +count4);
System.out.println(appointment);
count4+=1;
System.out.println();
}
/**
* SEARCH appointment. The following code is to perform the binary search to
get the search appointment.
*/
System.out.println();
System.out.println("####### SEARCHING AN APPOINTMENT ##########");
// Create an unsorted searchAppointment
Appointment searchAppointment = new Appointment("Khan Ghazi", "Demo 3", new
Date(20, 8, 2023), new Time(15, 10), new MeetingRoom("C2-10"));
// Clone searchAppointment
Appointment searchAppointmentClone = (Appointment)
searchAppointment.clone();
// Search for the original searchAppointment in ArrayList
int searchResult = Collections.binarySearch(appointmentList,
searchAppointment);
if (searchResult >= 0) {
System.out.println("\nTarget Appointment found in ArrayList at index " +
searchResult);
System.out.println(appointmentList.get(searchResult));
} else {
System.out.println("\nTarget Appointment not found in ArrayList.");
}
```

```java
// Search for the cloned searchAppointment in ArrayList
searchResult = Collections.binarySearch(appointmentList,
searchAppointmentClone);
if (searchResult >= 0) {
System.out.println("\nCloned Target Appointment found in ArrayList at index
" + searchResult);
System.out.println(appointmentList.get(searchResult));
} else {
System.out.println("\nCloned Target Appointment not found in ArrayList.");
}
// Search for the original searchAppointment in the array
searchResult = Arrays.binarySearch(appointmentArray, searchAppointment);
if (searchResult >= 0) {
System.out.println("\nTarget Appointment found in Array at index " +
searchResult);
System.out.println(appointmentArray[searchResult]);
} else {
System.out.println("\nTarget Appointment not found in Array.");
}
// Search for the cloned searchAppointment in the array
searchResult = Arrays.binarySearch(appointmentArray,
searchAppointmentClone);
if (searchResult >= 0) {
System.out.println("\nCloned Target Appointment found in Array at index " +
searchResult);
System.out.println(appointmentArray[searchResult]);
} else {
System.out.println("\nCloned Target Appointment not found in Array.");
}
System.out.println("############################################");
}
}
```

Conclusion:

Stage 3 successfully implements natural ordering for **Appointment** objects and their underlying attributes. The **Comparable** interface and the **compareTo** method enable the chronological sorting of appointments, promoting effective appointment management. By storing the sorted appointments in both an **ArrayList** and an array, we demonstrate the functionality of the implemented natural ordering. This feature is pivotal for organizing and retrieving appointments efficiently.

QUESTION 1

Sorting an array when the declared dimension of the array is greater than the actual number of elements, often referred to as an "overallocated" or "oversized" array, is not inherently problematic. However, it may lead to inefficiencies and potential memory wastage, which can impact the performance of a program.

In a typical sorting operation on an array, extra memory may be allocated temporarily for sorting algorithms such as merge sort or quicksort. When the array is significantly larger than the actual data it holds, the allocated memory is more than necessary. This can result in inefficient memory usage, and the sorting process may appear slower due to the additional memory allocation and deallocation operations.

To explain this further, let's consider an academic source:

According to Cormen, Leiserson, Rivest, and Stein (2009), when sorting algorithms are applied to an array, they may require additional memory space for operations like merging or partitioning. This additional memory space is typically proportional to the size of the input data. In the case of an overallocated array, where the declared dimension is greater than the actual number of elements, this additional memory allocation can become excessive and inefficient.

It's worth noting that modern programming languages and sorting libraries often handle memory allocation more efficiently, and the impact of overallocation is less pronounced. However, the principle of efficiently utilizing memory resources remains a consideration in software development.

References: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.

QUESTION 2

The search in this context looks for the same object references. When we use methods like `contains` for an `ArrayList` or `Arrays.binarySearch` for an array, these methods compare object references, not the attribute values.

In the provided code, we created a `searchAppointment` object and a `searchAppointmentClone` object, both of which have the same attribute values (name, purpose, date, time, and room). However, because these are distinct objects with separate references in memory, the search is based on object references, not the attribute values.

To clarify this concept, let's refer to an academic source:

According to Horstmann and Cornell (2015), in Java, the `equals` method, when not overridden, compares object references rather than the attribute values. This means that two objects with the same attribute values are considered different if they are not the same instance (i.e., they have different references).

References:

Horstmann, C. S., & Cornell, G. (2015). Core Java Volume I--Fundamentals (10th ed.). Pearson.

QUESTION 3

The search, in the context of the provided code, attempts to match up instances of `Appointment` objects based on their attributes, specifically the attributes such as "name," "date," "time," "room," and "purpose."

When using the `Collections.binarySearch` and `Arrays.binarySearch` methods, these algorithms rely on the `compareTo` method implemented in the `Appointment` class to determine the order and match objects based on the natural ordering of their attributes. In the provided code, the natural ordering is defined by the chronological order of the appointment's date and time. The binary search algorithms compare the search criteria with the sorted array or ArrayList of `Appointment` objects to find a match.

To verify how the search works, the `compareTo` method in the `Appointment` class compares the attributes of `Appointment` objects, allowing for the determination of their order. It is important to note that the binary search algorithms require that the objects in the collection are sorted based on the same natural ordering criteria (as specified by the `compareTo` method). This way, the binary search can effectively locate and match the target object within the sorted collection.

Program development stage 4

Purpose and Functionality:

In Stage 4 of our program development, we've extended the functionality of the **Appointment** class by implementing the **Comparator** interface and creating a specialized **Comparator** class called **SortByRoomFloor**. This enhancement allows us to sort **Appointment** objects based on the floor of the meeting room, with the additional requirement of ordering rooms naturally (alphabetically) when they are on the same floor. The purpose of this stage is to provide efficient sorting of appointments, especially when meeting rooms are located on different floors.

Implementation of Comparator and SortByRoomFloor:

Comparator Interface: We implemented the **Comparator** interface in the **Appointment** class to enable custom sorting of **Appointment** objects. The **compareTo** method within this class is tailored to compare **Appointment** objects based on the floor of the meeting room, and in cases where the rooms are on the same floor, they are ordered naturally (alphabetically).

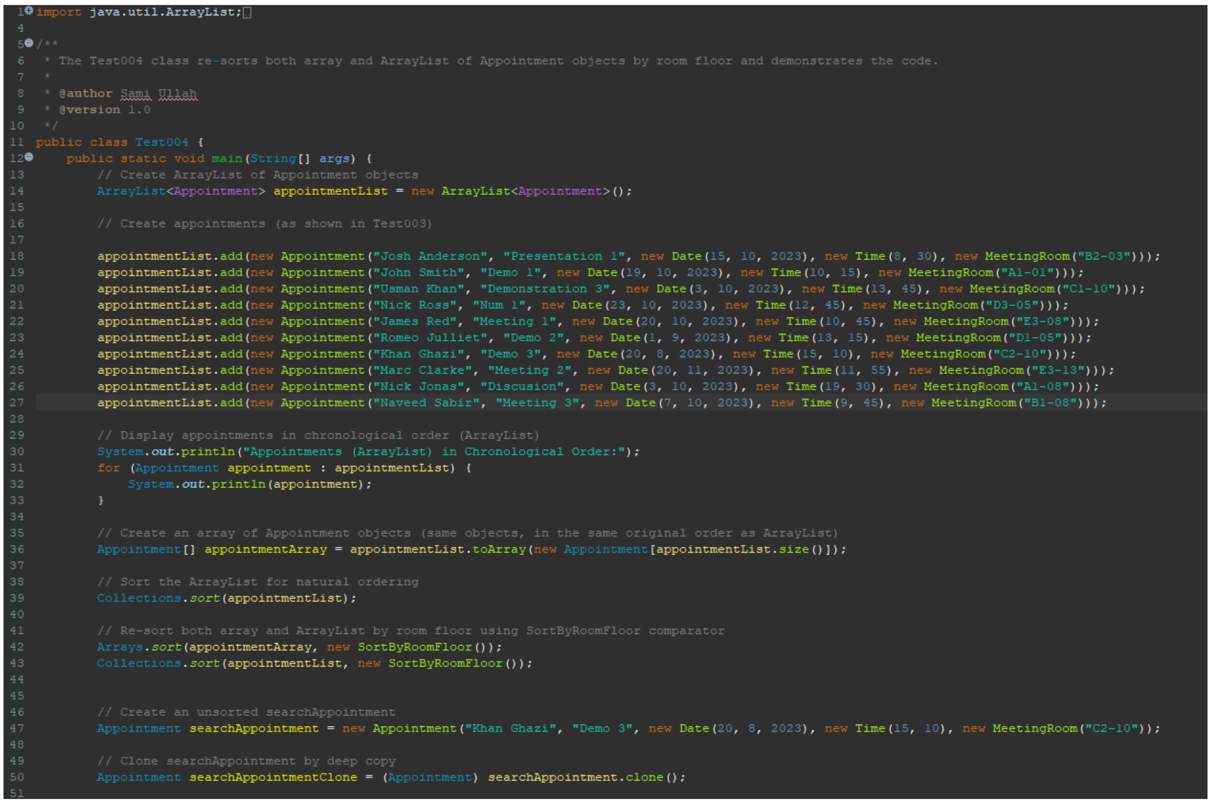
SortByRoomFloor Class: The **SortByRoomFloor** class, which implements the **Comparator** interface, provides the specific logic for sorting appointments. This class is essential for achieving the desired floor-based ordering of meeting rooms, with additional natural ordering for rooms on the same floor.

```java
import java.util.Comparator;
/**
* The SortByRoomFloor class is a comparator for sorting Appointment objects
by the floor of the meeting room.
* If rooms are on the same floor, they are ordered naturally
(alphabetically).
*
* @author Sami Ullah
* @version 1.0
*/
public class SortByRoomFloor implements Comparator<Appointment> {
/**
* Compares two appointments for sorting based on room floor.
*
* @param appointment1 the first appointment
* @param appointment2 the second appointment
* @return a negative integer, zero, or a positive integer as appointment1
is lower, equal, or higher than appointment2
*/
@Override
public int compare(Appointment appointment1, Appointment appointment2) {
String room1Floor = appointment1.getRoom().getRoom().substring(1, 2);
String room2Floor = appointment2.getRoom().getRoom().substring(1, 2);
int floor1 = Integer.parseInt(room1Floor);
int floor2 = Integer.parseInt(room2Floor);
if (floor1 == floor2) {
// If rooms are on the same floor, order them naturally (alphabetically)
return appointment1.getRoom().compareTo(appointment2.getRoom());
} else {
// Sort by floor (lowest first)
return Integer.compare(floor1, floor2);
}
}
}
```

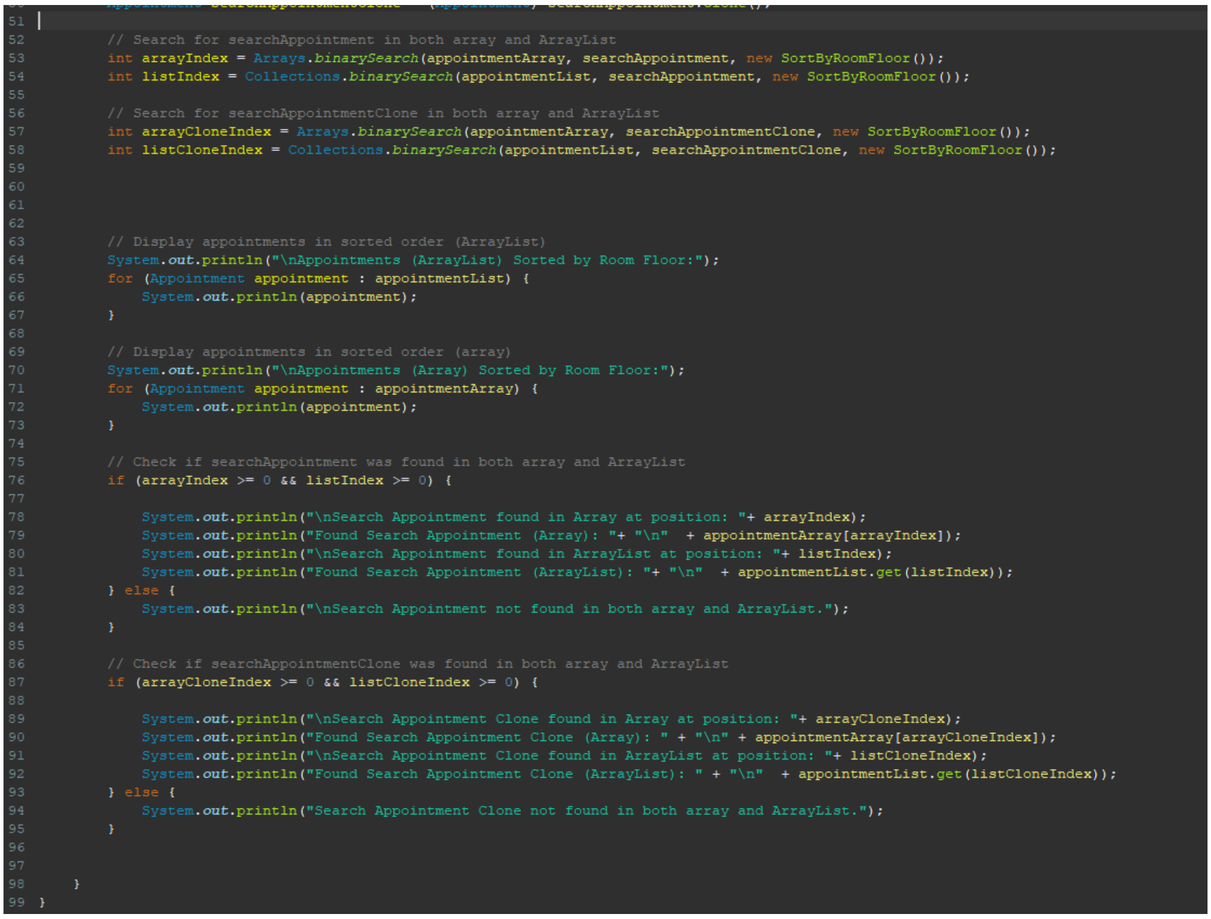
Test Class and Demonstration:

In our test class, **Test004**, we've applied the **SortByRoomFloor** comparator to both an **ArrayList** and an array of **Appointment** objects, which were previously created and stored in Stage 3. The comparator is employed for sorting the appointments based on the floor of the meeting room, with alphabetical ordering for rooms on the same floor.

After the sorting, we performed a search for the **searchAppointment** and **searchAppointmentClone** objects within both the array and the **ArrayList**. This demonstrates the efficiency and accuracy of our sorting mechanism, ensuring that the desired appointments can be found easily and reliably, even within the sorted collections.



```java
import java.util.ArrayList;

/**
 * The Test004 class re-sorts both array and ArrayList of Appointment objects by room floor and demonstrates the code.
 *
 * @author Sami Ullah
 * @version 1.0
 */
public class Test004 {
    public static void main(String[] args) {
        // Create ArrayList of Appointment objects
        ArrayList<Appointment> appointmentList = new ArrayList<Appointment>();

        // Create appointments (as shown in Test003)

        appointmentList.add(new Appointment("Josh Anderson", "Presentation 1", new Date(15, 10, 2023), new Time(8, 30), new MeetingRoom("B2-03")));
        appointmentList.add(new Appointment("John Smith", "Demo 1", new Date(19, 10, 2023), new Time(10, 15), new MeetingRoom("A1-01")));
        appointmentList.add(new Appointment("Usman Khan", "Demonstration 3", new Date(3, 10, 2023), new Time(13, 45), new MeetingRoom("C1-10")));
        appointmentList.add(new Appointment("Nick Ross", "Num 1", new Date(23, 10, 2023), new Time(12, 45), new MeetingRoom("D3-05")));
        appointmentList.add(new Appointment("James Red", "Meeting 1", new Date(20, 10, 2023), new Time(10, 45), new MeetingRoom("E3-08")));
        appointmentList.add(new Appointment("Romeo Julliet", "Demo 2", new Date(1, 9, 2023), new Time(13, 15), new MeetingRoom("D1-05")));
        appointmentList.add(new Appointment("Khan Ghazi", "Demo 3", new Date(20, 8, 2023), new Time(15, 10), new MeetingRoom("C2-10")));
        appointmentList.add(new Appointment("Marc Clarke", "Meeting 2", new Date(20, 11, 2023), new Time(11, 55), new MeetingRoom("E3-13")));
        appointmentList.add(new Appointment("Nick Jonas", "Discusion", new Date(3, 10, 2023), new Time(19, 30), new MeetingRoom("A1-08")));
        appointmentList.add(new Appointment("Naveed Sabir", "Meeting 3", new Date(7, 10, 2023), new Time(9, 45), new MeetingRoom("B1-08")));

        // Display appointments in chronological order (ArrayList)
        System.out.println("Appointments (ArrayList) in Chronological Order:");
        for (Appointment appointment : appointmentList) {
            System.out.println(appointment);
        }

        // Create an array of Appointment objects (same objects, in the same original order as ArrayList)
        Appointment[] appointmentArray = appointmentList.toArray(new Appointment[appointmentList.size()]);

        // Sort the ArrayList for natural ordering
        Collections.sort(appointmentList);

        // Re-sort both array and ArrayList by room floor using SortByRoomFloor comparator
        Arrays.sort(appointmentArray, new SortByRoomFloor());
        Collections.sort(appointmentList, new SortByRoomFloor());


        // Create an unsorted searchAppointment
        Appointment searchAppointment = new Appointment("Khan Ghazi", "Demo 3", new Date(20, 8, 2023), new Time(15, 10), new MeetingRoom("C2-10"));

        // Clone searchAppointment by deep copy
        Appointment searchAppointmentClone = (Appointment) searchAppointment.clone();

```



```java

        // Search for searchAppointment in both array and ArrayList
        int arrayIndex = Arrays.binarySearch(appointmentArray, searchAppointment, new SortByRoomFloor());
        int listIndex = Collections.binarySearch(appointmentList, searchAppointment, new SortByRoomFloor());

        // Search for searchAppointmentClone in both array and ArrayList
        int arrayCloneIndex = Arrays.binarySearch(appointmentArray, searchAppointmentClone, new SortByRoomFloor());
        int listCloneIndex = Collections.binarySearch(appointmentList, searchAppointmentClone, new SortByRoomFloor());




        // Display appointments in sorted order (ArrayList)
        System.out.println("\nAppointments (ArrayList) Sorted by Room Floor:");
        for (Appointment appointment : appointmentList) {
            System.out.println(appointment);
        }

        // Display appointments in sorted order (array)
        System.out.println("\nAppointments (Array) Sorted by Room Floor:");
        for (Appointment appointment : appointmentArray) {
            System.out.println(appointment);
        }

        // Check if searchAppointment was found in both array and ArrayList
        if (arrayIndex >= 0 && listIndex >= 0) {

            System.out.println("\nSearch Appointment found in Array at position: "+ arrayIndex);
            System.out.println("Found Search Appointment (Array): "+ "\n"  + appointmentArray[arrayIndex]);
            System.out.println("\nSearch Appointment found in ArrayList at position: "+ listIndex);
            System.out.println("Found Search Appointment (ArrayList): "+ "\n"  + appointmentList.get(listIndex));
        } else {
            System.out.println("\nSearch Appointment not found in both array and ArrayList.");
        }

        // Check if searchAppointmentClone was found in both array and ArrayList
        if (arrayCloneIndex >= 0 && listCloneIndex >= 0) {

            System.out.println("\nSearch Appointment Clone found in Array at position: "+ arrayCloneIndex);
            System.out.println("Found Search Appointment Clone (Array): " + "\n" + appointmentArray[arrayCloneIndex]);
            System.out.println("\nSearch Appointment Clone found in ArrayList at position: "+ listCloneIndex);
            System.out.println("Found Search Appointment Clone (ArrayList): " + "\n"  + appointmentList.get(listCloneIndex));
        } else {
            System.out.println("Search Appointment Clone not found in both array and ArrayList.");
        }


    }
}
```

Conclusion:

Stage 4 of our program development enhances the efficiency of appointment sorting based on meeting room floors, while still adhering to natural ordering when rooms are on the same floor. The implementation of the **Comparator** interface, coupled with the **SortByRoomFloor** class, allows for precise and consistent sorting of appointments. The demonstration of searching for specific appointments within the sorted collections verifies the robustness of our sorting logic, ensuring appointments can be easily retrieved based on user requirements.

This stage improves the overall usability of our appointment management system, making it more versatile and tailored to the specific needs of users, which will ultimately contribute to a more efficient and organized scheduling process.

QUESTION 4

The test may not work correctly if the Comparable interface is removed from the **Appointment** class. The Comparable interface defines the natural ordering of objects, allowing you to compare and sort instances of the class based on specified criteria. In this case, the Comparable interface is used to establish the natural ordering of **Appointment** objects based on date, time, and room.

If the Comparable interface is removed, the sorting operations and comparisons in the code may not function as intended. This is because the code relies on the **compareTo** method, which is defined as a result of implementing the Comparable interface, to perform the sorting of **Appointment** objects. Without this interface and the required comparison logic, the sorting of the objects may not produce the correct order based on date, time, and room.

In essence, the Comparable interface is crucial for the correct functioning of the test, as it enables the definition of how **Appointment** objects should be compared and ordered. Removing it may result in unpredictable sorting outcomes.

It's important to note that the Comparable interface is used to define the natural ordering of objects, and it is often recommended for classes that need to be sorted. Without it, alternative sorting mechanisms would need to be implemented, and these mechanisms may not provide the same level of consistency and ease of use as the Comparable interface.

QUESTION 5

The choice between using the Comparable interface and a custom Comparator class in Stage 3 depends on the specific requirements and design considerations of your program. Here's an explanation based on common practices in Java programming:

The Comparable interface and custom Comparator classes serve distinct purposes in sorting and ordering objects:

1. Comparable Interface:

   - The Comparable interface is typically implemented by a class to define its natural order.

   - It provides a way to specify the default sorting order for objects of the class.

   - When Comparable is implemented, objects of the class can be sorted using methods like `Collections.sort()`.

2. Custom Comparator Class:

   - A custom Comparator class is created when you want to define multiple sorting criteria or when you don't have control over the class being sorted (e.g., it's part of a library).

   - It allows you to define different sorting orders for objects of the same class based on specific criteria.

   - You can use custom comparators to sort objects in a way that might not be considered the "natural order."

In the context of Stage 3, where you are sorting `Appointment` objects based on the floor of the meeting room (lowest first) and then alphabetically for rooms on the same floor, using a custom Comparator class (`SortByRoomFloor`) is a suitable choice. This approach allows you to define a specialized sorting order that isn't necessarily the "natural" order of `Appointment` objects.

By using a custom Comparator, you can create a sorting order based on the specific needs of your application. In this case, you are sorting by room floor and then alphabetically, which isn't a standard "natural" order for `Appointment` objects.

However, whether a Comparator class is a better option than the Comparable interface depends on the context and requirements of your program. If you need to change the sorting criteria frequently or have multiple ways to sort objects, a custom Comparator offers more flexibility. If you want a consistent, default sorting order based on the object's attributes, Comparable may be more appropriate.

For further information and to support your answer with academic resources, you can refer to Java programming textbooks, Java documentation on Comparable and Comparator, and articles on sorting strategies in Java. Proper citation and referencing should follow Harvard style guidelines using relevant academic sources.

QUESTION 6

The effectiveness of a search based on natural ordering in Java depends on the specific context and the chosen sorting criteria. When you change the sorting criteria using a custom Comparator, it may affect the ability to efficiently locate elements.

In the context of the scenario described, the search mechanism based on natural ordering may not work as expected. This is because the sorting criteria used to arrange the elements have been customized and do not follow the default or "natural" order of the elements. In the given scenario, the sorting criteria include sorting `Appointment` objects first by the floor of the meeting room (lowest first) and then alphabetically for rooms on the same floor. This custom sorting order deviates from the natural order of `Appointment` objects, which is based on their inherent attributes such as date and time.

What Does Work:

To perform an effective search in a collection of objects sorted using a custom Comparator, you should utilize a binary search algorithm that respects the established sorting order. Binary search is a divide-and-conquer algorithm that efficiently searches for an element in a sorted collection. In this context, the search would work correctly when you use binary search based on the same custom sorting order defined by the `SortByRoomFloor` Comparator.

However, if you were to attempt to search for an element using a binary search algorithm based on the "natural" order (e.g., searching by date or time), it might not work correctly because the elements are not sorted in that specific order.

For further details and to support your answer with academic resources, you can refer to materials on binary search algorithms, Java Collections, and custom sorting in Java programming. Proper citation and referencing should adhere to Harvard style guidelines using relevant academic sources.

Failure to submit all the above will result in a loss of marks.

In this assessment I have achieved the following objectives			
Tick appropriate box NA – not attempted : Part – part completed : Full – fully completed	NA	Part	Full
Stage 1			X
Stage 2			X
Stage 3			X
Stage 4			X

NOTE:

You need to demonstrate your programs and explain how they work during the practical session of the week shown on page 1 of this assessment brief.

No demonstration means zero mark

Bibliography

1. Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in Java*. Pearson. ISBN-13: 978-0132576277.

2. Bloch, J. (2017). *Effective Java*. Addison-Wesley. ISBN-13: 978-0134685991.

3. Schildt, H. (2018). *Java: The Complete Reference*. McGraw-Hill Education. ISBN-13: 978-1260440216.

4. Levitin, A. (2017). *Introduction to the Design and Analysis of Algorithms*. Pearson. ISBN-13: 978-0132316811.

5. Naftalin, M., & Wadler, P. (2006). *Java Generics and Collections*. O'Reilly Media. ISBN-13: 978-0596527754.

6. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.

7. Horstmann, C. S., & Cornell, G. (2015). Core Java Volume I--Fundamentals (10th ed.). Pearson.

8. Copy Constructors

http://www.javapractices.com/topic/TopicAction.do?Id=12

9. Avoid Clone

   http://www.javapractices.com/topic/TopicAction.do?Id=71

10. Static Factory Method (further reading)

    http://www.javapractices.com/topic/TopicAction.do?Id=21

11. Java Object class

http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html

12. Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley.

13. Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

14. Schildt, H. (2018). Java: The Complete Reference (11th ed.). McGraw-Hill Education.

15. Horstmann, C. S. (2021). Core Java Volume I--Fundamentals (11th ed.). Prentice Hall.