NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT I FOR
Semester 2 AY2022/2023

CS2030S Programming Methodology II

March 2023 Time Allowed 90 minutes

INSTRUCTIONS TO CANDIDATES

1. This practical assessment consists of **one** question. The total mark is 20: 12 marks for design; 3 for style; 5 for correctness. Style and correctness are given on the condition that reasonable efforts have been made to solve the given tasks.
2. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
3. You should see the following in your home directory.
   - The files `Test1.java`, `Test2.java`, ... to `Test7.java` for testing your solution.
   - The file `TaskList.java` for you to improve upon.
   - The directories `inputs` and `outputs` contain the test inputs and outputs.
   - The directory `pristine` contains a copy of the original test cases and code for reference.
   - The file `Array.java` that implements the generic array `Array<T>`.
   - The files `checkstyle.sh`, `checkstyle.jar`, `cs2030_checks.xml`, and `test.sh` are given to check the style of your code and to test your code.
   - You may add new classes/interfaces as needed by the design.
4. Solve the programming tasks by editing `TaskList.java` and creating any necessary files. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
5. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
6. Write your student number on top of EVERY FILE you created or edited as part of the `@author` tag. Do not write your name.
7. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.
8. To run all the test cases, run `./test.sh`. You can also run the test individually. For instance, run `java Test1 < inputs/Test1.1.in` to execute `Test1` on input `inputs/Test1.1.in`.
9. To check the style of your code, run `./checkstyle.sh`. You can also check the style of individual file with `java -jar checkstyle.jar -c cs2030_checks.xml <FILENAME>`.

IMPORTANT: If the submitted classes or any of the new files you have added cannot be compiled, 0 marks will be given. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

You have been given a class called `TaskList` that implements a list of to-do tasks. The class reads a list of tasks from a file and provides several APIs to print the list, remind users about the tasks, mark a task as completed, and calculate the reward points for completing tasks.

The class `TaskList`, however, is written without following object-oriented principles. You are asked to re-write the class `TaskList` to adhere to the object-oriented principles you have learned. You may create new classes as needed.

Your revised `TaskList` must follow the same behavior as the given `TaskList` (a copy of which can be found in `pristine/TaskList.java` for your reference). We give an overview of the behavior of `TaskList` here. For details, please see `TaskList.java`.

Constructors. An instance of a `TaskList` can be created using the constructor that takes in a `String` (representing the name of a text file) as an argument. A constructor for `TaskList` may also take in no argument. In this case, it reads the input from the standard inputs.

```
TaskList list1 = new TaskList(filename); // read from file
TaskList list2 = new TaskList(); // read from standard input
```

Types of Tasks. The class can handle three different types of tasks.

- Type 0: A task that is without a deadline and can be completed any time.
- Type 1: A task that comes with a deadline.
- Type 2: A task that comes with a deadline and is delegated to an assignee to complete.

Note that we cannot assign a task without a deadline to an assignee.

Input File. The file to be loaded into `TaskList` through the constructor has the following format.

- The first line of the file contains a positive integer n, which is the number of tasks.
- The next n lines contain information about the tasks. Each of these n lines contains two or more fields, separated by a comma.
  - The first field is always an integer and indicates the types of tasks (0, 1, 2)
  - The second field is a string that describes the task.
  - The third field applies only if the task has a deadline. This field is a non-negative integer that indicates the number of days before the task is due.
  - The fourth field applies only if the task has a deadline and is assigned to someone else to complete. This field is a string that contains the name of the assignee.

You can assume that the given test data follows the input format correctly, with the exception that the first field might indicate an invalid task type (neither 0, 1, nor 2). An example input file `Sample.txt` is given to the right.

```
4
0,Finish Quiz
2,Setup Server,5,Foo
1,Email Ah Keong,2
1,Revise CS2030S,0
```

Listing Tasks. There are two methods provided to list the task, `printTaskDescriptions` and `printTaskDetails`.

The method `printTaskDescriptions` takes in no arguments and returns nothing. It prints the description of each task enumerated in the same order as they appear in the input files.

For example `new TaskList("Sample.txt").printTaskDescriptions()` would print

```
0 Finish Quiz
1 Setup Server
2 Email Ah Keong
3 Revise CS2030S
```

The method `printTaskDetails` takes in no arguments and returns nothing. It prints the detailed information of each task enumerated in the same order as they appear in the input files, including the due date (if any), assignee (if any), and whether the task is completed or not.

For example `new TaskList("Sample.txt").printTaskDetails()` would print

```
0 [ ] Finish Quiz
1 [ ] Setup Server | Due in 5 days | Assigned to Foo
2 [ ] Email Ah Keong | Due in 2 days
3 [ ] Revise CS2030S | Due in 0 days
```

Completing Tasks. To complete a task, we can call `completeTask` with the task index as an argument. For instance, to complete Task 2 "Email Ah Keong", we call `completeTask(2)`. After we execute:

```
TaskList list = new TaskList("Sample.txt");
list.complete(2);
list.printTaskDetails();
```

the following will be printed

```
0 [ ] Finish Quiz
1 [ ] Setup Server | Due in 5 days | Assigned to Foo
2 [X] Email Ah Keong | Due in 2 days
3 [ ] Revise CS2030S | Due in 0 days
```

Completing a task that has been completed has no effect.

Tasks Due Today. We can also call the method `printDueToday` to print the details of all tasks that are due today, i.e., in 0 days.

Executing this snippet

```
TaskList list = new TaskList("Sample.txt");
list.printDueToday();
```

would cause the following to be printed:

```
3 [ ] Revise CS2030S | Due in 0 days
```

Reminders. The class `TaskList` provides a method `remindAll` that can go through all incomplete tasks with deadlines and print a reminder.

If a task has an assignee, it prints a string with the format "Sending a reminder to complete DESCRIPTION to ASSIGNEE", replacing DESCRIPTION and ASSIGNEE with the actual description and assignee of the task.

If the task has no assignee, it reminds the owner of the task by printing "The task DESCRIPTION is due in DUE_IN days", replacing DESCRIPTION and DUE_IN with the actual description and the number of days the task is due in.

For example, executing this snippet

```
TaskList list = new TaskList("Sample.txt");
list.complete(2);
list.remindAll();
```

causes the following to be printed:

```
Sending a reminder to complete "Setup Server" to Foo
The task "Revise CS2030S" is due in 0 days
```

Reward Points. Users can receive reward points for completing tasks before their deadlines. If a task is completed k days before its due, k points would be rewarded. This reward point is given even for tasks that are delegated to assignees. The method `getRewardPoints` returns the number of reward points accumulated after completing the tasks.

For example, executing this snippet:

```
TaskList list = new TaskList("Sample.txt");
list.completeTask(2);
list.completeTask(0);
list.completeTask(1);
list.getRewardPoints(); // return 7
```

would return 7 since completing "Setup Server" yields 5 points and completing "Email Ah Keong" yields 2 points. The task "Finish Quiz" has no deadline and so completing it does not contribute to the reward points.

Handling Errors. You can assume that the input file format is valid. However, the type of tasks in the input might be an integer that is out of range, i.e., neither 0, 1, nor 2.

When the class `TaskList` loads an input file with such an error, it would print an error message with the invalid task type. For example, if the task type is given as 4, it would print: `Invalid task type in input: 4` and abort the loading of the tasks.

Your Tasks

Task 1: Rewrite this program using OOP principles

You should read through the file `TaskList.java` to understand what it is doing. The given implementation applies minimal OO principles, your task in this exam is to rewrite `TaskList.java`, including adding new classes to apply the OO principles you learned.

To achieve this, create a new class called `Task` to encapsulate the relevant attributes and methods. Create subclasses of `Task` as necessary. Use polymorphism to simplify the code in `TaskList` and make your code extensible to possible new task types in the future. Make sure all OO principles, including LSP, tell-don't-ask, information hiding, are adhered to.

Solution: Task.java

```
abstract class Task {
  private String description;
  private boolean isCompleted = false;

  public Task(String description) {
    this.description = description;
  }

  public void complete() {
    this.isCompleted = true;
  }

  public boolean isCompleted() {
    return this.isCompleted;
  }

  public abstract boolean isDueToday();
  public abstract int getRewardPoints();
  public abstract void remind();

  public String toPrettyString() {
```

```
    if (this.isCompleted) {
      return "[X] " + this;
    } else {
      return "[ ] " + this;
    }
  }

  @Override
  public String toString() {
    return this.description;
  }
}
```

AnytimeTask.java

```
public class AnytimeTask extends Task {
  public AnytimeTask(String description) {
    super(description);
  }

  @Override
  public boolean isDueToday() {
    return false;
  }

  @Override
  public int getRewardPoints() {
    return 0;
  }

  @Override
  public void remind() {
    // do nothing
  }
}
```

AssignableTask.java

```
public class AssignableTask extends DeadlineTask {
  private String assignee;

  public AssignableTask(String taskText, int dueInDays, String assignee) {
    super(taskText, dueInDays);
    this.assignee = assignee;
  }

  @Override
  public void remind() {
    System.out.println("Sending a reminder to complete \"" +
        this + "\" to " + assignee);
  }

  @Override
  public String toPrettyString() {
    return super.toPrettyString() + " | Assigned to " + assignee;
```

```
  }
}
```

DeadlineTask.java

```
class DeadlineTask extends Task {
  private int dueInDays;
  private int points;

  public DeadlineTask(String description, int dueInDays) {
    super(description);
    this.dueInDays = dueInDays;
  }

  @Override
  public void complete() {
    super.complete();
    this.points += this.dueInDays;
  }

  @Override
  public int getRewardPoints() {
    return this.points;
  }

  @Override
  public boolean isDueToday() {
    return dueInDays == 0;
  }

  @Override
  public void remind() {
    System.out.println("The task \"" + this + "\" is due in "
        + dueInDays + " days");
  }

  @Override
  public String toPrettyString() {
    return super.toPrettyString() + " | Due in " + dueInDays + " days";
  }
}
```

Task 2: Implement Exception handling
In the current implementation, the methods `createTask` and `loadTasks` return a boolean flag to indicate if the method is successful or not, and stored the invalid task type in an attribute `errorMsg`. When this error is encountered in `createTask`, it updates the `errorMsg`. The error is handled in the constructor of `TaskList`.

With exception handling, we can remove the need to return a boolean flag to indicate if the method is successful or not, and remove the need to maintain the attribute `errorMsg`.

To achieve this, create a new checked exception called `WrongTaskTypeException` and throws this exception from `createTask` when an error is encountered. Catch and handle this exception in the constructor of `TaskList`.

In your revised code, `createTask` and `loadTasks` must return `void` instead. The error message must no longer be stored in `TaskList` as an attribute.

Reminder: all checked exceptions are a subclass of `java.lang.Exception`. The class `Exception` has the following constructor:

```
Exception(String msg)
```

that constructs a new exception with the specified detail message `msg`. The message can be retrieved by the `getMessage()` method, which returns the message as a `String`.

Solution: WrongTaskTypeException.java

```
public class WrongTaskTypeException extends Exception {
  public WrongTaskTypeException(String exceptionMessage) {
    super(exceptionMessage);
  }
}
```

TaskList.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class TaskList {
  private Array<Task> array;

  private static final int TODO = 0;
  private static final int DEADLINE = 1;
  private static final int ASSIGNABLE = 2;

  public TaskList() {
    this(new Scanner(System.in));
  }

  public TaskList(String filename) throws FileNotFoundException {
    this(new Scanner(new File(filename)));
  }

  private TaskList(Scanner sc) {
    try {
      loadTasks(sc);
    } catch (WrongTaskTypeException e) {
      System.out.println(e.getMessage());
    } finally {
      sc.close();
    }
  }

  private void loadTasks(Scanner sc) throws WrongTaskTypeException {
    int numOfTasks = Integer.parseInt(sc.nextLine().trim());
    array = new Array<Task>(numOfTasks);
    int i = 0;
    while (sc.hasNext()) {
      String text = sc.nextLine().trim();
      String arguments[] = text.split(",");
      array.set(i, createTask(arguments));
```

```
      i = i + 1;
    }
  }

  private Task createTask(String[] args) throws WrongTaskTypeException {
    String description = args[1];
    int type = Integer.parseInt(args[0]);
    Task t = null;
    if (type == TaskList.TODO) {
      t = new AnytimeTask(description);

    } else if (type == TaskList.DEADLINE) {
      int dueInDays = Integer.parseInt(args[2]);
      t = new DeadlineTask(description, dueInDays);

    } else if (type == TaskList.ASSIGNABLE) {
      int dueInDays = Integer.parseInt(args[2]);
      String assignee = args[3];
      t = new AssignableTask(description, dueInDays, assignee);

    } else {
      throw new WrongTaskTypeException("Invalid task type in input: " + type);
    }
    return t;
  }

  @Override
  public String toString() {
    String s = "";
    int i = 0;
    for (i = 0; i < array.length(); i++) {
      s = s + "\n" + array.get(i);
    }
    return s;
  }

  public void printDueToday() {
    for (int i = 0; i < array.length(); i++) {
      Task task = array.get(i);
      if (task.isDueToday()) {
        System.out.println(i + " " + task.toPrettyString());
      }
    }
  }

  public int getRewardPoints() {
    int sum = 0;
    for (int i = 0; i < array.length(); i++) {
      Task task = array.get(i);
      sum += task.getRewardPoints();
    }
    return sum;
  }

  public void printTaskDescriptions() {
```

```
    for (int i = 0; i < array.length(); i++) {
      Task task = array.get(i);
      System.out.println(i + " " + task);
    }
  }

  public void printTaskDetails() {
    for (int i = 0; i < array.length(); i++) {
      Task task = array.get(i);
      System.out.println(i + " " + task.toPrettyString());
    }
  }

  public void remindAll() {
    for (int i = 0; i < array.length(); i++) {
      Task task = array.get(i);
      if (!task.isCompleted()) {
        task.remind();
      }
    }
  }

  public void completeTask(int index) {
    array.get(index).complete();
  }
}
```

END OF PAPER

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT I FOR
Semester 2 AY2022/2023

CS2030S Programming Methodology II

March 2023 Time Allowed 90 minutes

INSTRUCTIONS TO CANDIDATES

1. This practical assessment consists of **one** question. The total mark is 20: 12 marks for design; 3 for style; 5 for correctness. Style and correctness are given on the condition that reasonable efforts have been made to solve the given tasks.
2. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
3. You should see the following in your home directory.
   - The files `Test1.java`, `Test2.java`, ... to `Test7.java` for testing your solution.
   - The file `DayCalendar.java` for you to improve upon.
   - The directories `inputs` and `outputs` contain the test inputs and outputs.
   - The directory `pristine` contains a copy of the original test cases and code for reference.
   - The file `Array.java` that implements the generic array `Array<T>`.
   - The files `checkstyle.sh`, `checkstyle.jar`, `cs2030_check.xml`, and `test.sh` are given to check the style of your code and to test your code.
   - You may add new classes/interfaces as needed by the design.
4. Solve the programming tasks by editing `DayCalendar.java` and creating any necessary files. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
5. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
6. Write your student number on top of EVERY FILE you created or edited as part of the `@author` tag. Do not write your name.
7. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.
8. To run all the test cases, run `./test.sh`. You can also run the test individually. For instance, run `java Test1 < inputs/Test1.1.in` to execute `Test1` on input `inputs/Test1.1.in`.
9. To check the style of your code, run `./checkstyle.sh`. You can also check the style of individual file with `java -jar checkstyle.jar -c cs2030_checks.xml <FILENAME>`.

IMPORTANT: If the submitted classes or any of the new files you have added cannot be compiled, 0 marks will be given. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

You have been given a class called `DayCalendar` that implements a one-day event calendar. The class reads a list of events from a file and provides several APIs to print the events, remind users about upcoming events, cancel a meeting, and calculate the busy period for the day.

The class `DayCalendar`, however, is written without following object-oriented principles. You are asked to re-write the class `DayCalendar` to adhere to the object-oriented principles you have learned. You may create new classes as needed.

Your revised `DayCalendar` must follow the same behavior as the given `DayCalendar` (a copy of which can be found in `pristine/DayCalendar.java` for your reference). We give an overview of the behavior of `DayCalendar` here. For details, please see `DayCalendar.java`.

Constructors. An instance of a `DayCalendar` can be created using the constructor that takes in a `String` (representing the name of a text file) as an argument. A constructor for `DayCalendar` may also take in no argument. In this case, it reads the input from the standard inputs.

```
DayCalendar cal1 = new DayCalendar(filename); // read from file
DayCalendar cal2 = new DayCalendar(); // read from standard input
```

Types of Events. The class can handle three different types of events.

- Type 0: A birthday, which is an all-day event without a starting time and an ending time.
- Type 1: A lesson, which is a timed event, i.e., has a starting and ending time.
- Type 2: A meeting, which is also a timed event.

A birthday event is associated with the name of a person whose birthday is today; A meeting event is associated with the name of the person to meet with.

Input File. The file to be loaded into `DayCalendar` through the constructor has the following format.

- The first line of the text contains a positive integer n, which is the number of events.
- The next n lines contain information about the events. Each of these n lines contains two or more fields, separated by a comma.
  - The first field is always an integer and indicates the types of events (0, 1, 2)
  - The second field is a string that describes the event. For a birthday event, this is the name of the person whose birthday is today.
  - The third and fourth fields are positive integers that represent the starting time and ending time, in hours. These two fields only apply to timed events. The ending time is always greater than or equal to the starting time.
  - The fifth field applies to meeting events. This field is a string that contains the name of the person to meet with.

You can assume that the given test data follows the input format correctly.

```
5
1,CS2030S,12,14
0,Ah Huat
2,Discuss Project,11,12,Ahmad
1,MA2101,8,10
2,Breakfast,10,11,Devi
```

Listing Events. There are two methods provided to list the events, `printEventDescriptions` and `printEventDetails`.

The method `printEventDescriptions` takes in no arguments and returns nothing. It prints the description of each non-cancelled event enumerated in the same order as they appear in the input files.

For example `new DayCalendar("Sample.txt").printEventDescriptions()` would print

```
0 CS2030S
1 Birthday (Ah Huat)
2 Discuss Project
3 MA2101
4 Breakfast
```

Note that for bithday events, the name of the person whose birthday is today is also printed in parenthesis.

The method `printEventDetails` takes in no arguments and returns nothing. It prints the non-cancelled detailed information of each event enumerated in the same order as they appear in the input files, including the starting time (if any), ending time (if any), and the person associated with the event (if any).

For example `new DayCalendar("Sample.txt").printEventDetails()` would print

```
0 CS2030S | 12 - 14
1 Birthday (Ah Huat)
2 Discuss Project | 11 - 12 | Meet with Ahmad
3 MA2101 | 8 - 10
4 Breakfast | 10 - 11 | Meet with Devi
```

Cancelling Events. To cancel an event, we can call `cancelEvent` with the event index as an argument. For instance, to cancel Event 2, the meeting with Ahmad, we call `cancelEvent(2)`. After we execute:

```
DayCalendar cal = new DayCalendar("Sample.txt");
cal.cancelEvent(2);
cal.printEventDetails();
```

The following will be printed

```
0 CS2030S | 12 - 14
1 Birthday (Ah Huat)
3 MA2101 | 8 - 10
4 Breakfast | 10 - 11 | Meet with Devi
```

Only a meeting can be cancelled. Trying to cancel a birthday event or a lesson would cause an error message to be printed. For example, if we run

```
DayCalendar cal = new DayCalendar("Sample.txt");
cal.cancelEvent(0);
cal.cancelEvent(1);
cal.printEventDetails();
```

the following will be printed

```
Unable to cancel event: CS2030S
Unable to cancel event: Birthday (Ah Huat)
0 CS2030S | 12 - 14
1 Birthday (Ah Huat)
2 Discuss Project | 11 - 12 | Meet with Ahmad
3 MA2101 | 8 - 10
4 Breakfast | 10 - 11 | Meet with Devi
```

Trying to cancel a cancelled event has no effect.

Reminders. The class `DayCalendar` provides a method `remind` that lists all events that have not started yet.

The method `remind` takes in the current time (in hour) and list all events that have not been cancelled, where the starting time is greater than or equal to the given time. For example, executing this snippet

```
DayCalendar cal = new DayCalendar("Sample.txt");
cal.remind(10);
```

causes the following to be printed:

```
0 CS2030S | 12 - 14
2 Discuss Project | 11 - 12 | Meet with Ahmad
4 Breakfast | 10 - 11 | Meet with Devi
```

Executing this snippet

```
DayCalendar cal = new DayCalendar("Sample.txt");
cal.cancelEvent(2);
cal.remind(10);
```

causes the following to be printed instead.

```
0 CS2030S | 12 - 14
4 Breakfast | 10 - 11 | Meet with Devi
```

Busy Period. Users can find out how many hours they are busy with lessons and meetings.

The method `getBusyPeriod` returns the total number of hours that the user is busy. Events that have been cancelled should not contributes towards the busy period.

For example, executing this snippet

```
DayCalendar cal = new DayCalendar("Sample.txt");
cal.cancelEvent(2);
cal.getBusyPeriod(); // return 5
```

would return 6 since taking CS2030S and MA2101 took 4 hours and the meeting with Devi took 1 hour. The meeting with Ahmad has been cancelled.

Your Tasks

Task 1: Rewrite this program using OOP principles

You should read through the file `DayCalendar.java` to understand what it is doing. The given implementation applies minimal OO principles, your task in this exam is to rewrite `DayCalendar.java`, including adding new classes to apply the OO principles you learned.

To achieve this, create a new class called `Event` to encapsulate the relevant attributes and methods. Create subclasses of `Event` as necessary. Use polymorphism to simplify the code in `DayCalendar` and make your code extensible to possible new event types in the future. Make sure all OO principles, including LSP, tell-don't-ask, information hiding, are adhered to.

Solution:

Event.java

```
abstract class Event {
  private String description;
  private boolean isCancelled;

  public Event(String description) {
    this.description = description;
    this.isCancelled = false;
  }

  public abstract int getBusyPeriod();
  public abstract String toPrettyString();
  public abstract boolean isPast(int time);

  public void cancel() throws IllegalCancellationException {
```

```
    this.isCancelled = true;
  }

  public boolean isCancelled() {
    return this.isCancelled;
  }

  @Override
  public String toString() {
    return description;
  }
}
```

TimedEvent.java

```
abstract class TimedEvent extends Event {
  private int startTime;
  private int endTime;

  public TimedEvent(String description, int startTime, int endTime) {
    super(description);
    this.startTime = startTime;
    this.endTime = endTime;
  }

  @Override
  public int getBusyPeriod() {
    if (!this.isCancelled()) {
      return endTime - startTime;
    }
    return 0;
  }

  @Override
  public String toPrettyString() {
    return super.toString() + " | " + startTime + " - " + endTime;
  }

  @Override
  public boolean isPast(int time) {
    return startTime < time;
  }
}
```

BirthdayEvent.java

```
class BirthdayEvent extends Event {
  private String who;

  public BirthdayEvent(String who) {
    super("Birthday");
    this.who = who;
  }

  @Override
```

```
  public int getBusyPeriod() {
    return 0;
  }

  @Override
  public String toPrettyString() {
    return "Birthday (" + who + ")";
  }

  @Override
  public String toString() {
    return "Birthday (" + who + ")";
  }

  @Override
  public boolean isPast(int time) {
    return true;
  }

  @Override
  public void cancel() throws IllegalCancellationException {
    throw new IllegalCancellationException(this);
  }
}
```

LessonEvent.java

```
class LessonEvent extends TimedEvent {
  public LessonEvent(String description, int startTime, int endTime) {
    super(description, startTime, endTime);
  }

  @Override
  public void cancel() throws IllegalCancellationException {
    throw new IllegalCancellationException(this);
  }
}
```

MeetingEvent.java

```
class MeetingEvent extends TimedEvent {
  private String withWho;

  public MeetingEvent(String description, int startTime, int endTime, String withWho) {
    super(description, startTime, endTime);
    this.withWho = withWho;
  }

  @Override
  public String toPrettyString() {
    return super.toPrettyString() + " | Meet with " + withWho;
  }
}
```

Task 2: Implement Exception Handling
To handle the error where user tries to cancel a birthday event or a lesson event, create a new checked exception called `IllegalCancellationException` and throws this exception from `Event` (or its subclasses, if any) when appropriate. Catch and handle this exception in the method `cancelEvent` of `DayCalendar`.

Reminder: all checked exceptions are a subclass of `java.lang.Exception`. The class `Exception` has the following constructor:

```
Exception(String msg)
```

that constructs a new exception with the specified detail message `msg`. The message can be retrieved by the `getMessage()` method, which returns the message as a `String`.

Solution: IllegalCancellationException.java

```
class IllegalCancellationException extends Exception {
  private Event e;
  public IllegalCancellationException(Event e) {
    this.e = e;
  }

  @Override
  public String toString() {
    return "Unable to cancel event: " + e;
  }
}
```

DayCalendar.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class DayCalendar {
  private Array<Event> array;

  private static final int BIRTHDAY = 0;
  private static final int LESSON = 1;
  private static final int MEETING = 2;

  public DayCalendar() {
    this(new Scanner(System.in));
  }

  public DayCalendar(String filename) throws FileNotFoundException {
    this(new Scanner(new File(filename)));
  }

  private DayCalendar(Scanner sc) {
    loadEvents(sc);
  }

  private void loadEvents(Scanner sc) {
    int numOfEvents = Integer.parseInt(sc.nextLine().trim());
    array = new Array<Event>(numOfEvents);
```

```
    int i = 0;
    while (sc.hasNext()) {
      String text = sc.nextLine().trim();
      String arguments[] = text.split(",");
      array.set(i, createEvent(arguments));
      i = i + 1;
    }
  }

  private Event createEvent(String[] args) {
    int type = Integer.parseInt(args[0]);
    String description = args[1];
    Event t = null;

    if (type == DayCalendar.LESSON) {
      int start = Integer.parseInt(args[2]);
      int end = Integer.parseInt(args[3]);
      t = new LessonEvent(description, start, end);

    } else if (type == DayCalendar.BIRTHDAY) {
      t = new BirthdayEvent(description);

    } else if (type == DayCalendar.MEETING) {
      int start = Integer.parseInt(args[2]);
      int end = Integer.parseInt(args[3]);
      String meetingWith = args[4];
      t = new MeetingEvent(description, start, end, meetingWith);
    }
    return t;
  }

  @Override
  public String toString() {
    String s = "";
    int i = 0;
    for (i = 0; i < array.length(); i++) {
      s = s + "\n" + array.get(i);
    }
    return s;
  }

  public int getBusyPeriod() {
    int sum = 0;
    for (int i = 0; i < array.length(); i++) {
      Event event = array.get(i);
      sum += event.getBusyPeriod();
    }
    return sum;
  }

  public void printEventDescriptions() {
    for (int i = 0; i < array.length(); i++) {
      Event event = array.get(i);
      if (!event.isCancelled()) {
        System.out.println(i + " " + event);
```

```
      }
    }
  }

  public void printEventDetails() {
    for (int i = 0; i < array.length(); i++) {
      Event event = array.get(i);
      if (!event.isCancelled()) {
        System.out.println(i + " " + event.toPrettyString());
      }
    }
  }

  public void remind(int time) {
    for (int i = 0; i < array.length(); i++) {
      Event event = array.get(i);
      if (!event.isCancelled()) {
        if (!event.isPast(time)) {
          System.out.println(i + " " + event.toPrettyString());
        }
      }
    }
  }

  public void cancelEvent(int index) {
    try {
      array.get(index).cancel();
    } catch (IllegalCancellationException e) {
      System.out.println(e);
    }
  }
}
```

END OF PAPER

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT I FOR
Semester 2 AY2022/2023

CS2030S Programming Methodology II

March 2023 Time Allowed 90 minutes

INSTRUCTIONS TO CANDIDATES

1. This practical assessment consists of **one** question. The total mark is 20: 12 marks for design; 3 for style; 5 for correctness. Style and correctness are given on the condition that reasonable efforts have been made to solve the given tasks.
2. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
3. You should see the following in your home directory.
   - The files `Test1.java`, `Test2.java`, ... to `Test7.java` for testing your solution.
   - The file `CallHistory.java` for you to improve upon.
   - The directories `inputs` and `outputs` contain the test inputs and outputs.
   - The directory `pristine` contains a copy of the original test cases and code for reference.
   - The file `Array.java` that implements the generic array `Array<T>`.
   - The files `checkstyle.sh`, `checkstyle.jar`, `cs2030_check.xml`, and `test.sh` are given to check the style of your code and to test your code.
   - You may add new classes/interfaces as needed by the design.
4. Solve the programming tasks by editing `CallHistory.java` and creating any necessary files. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
5. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
6. Write your student number on top of EVERY FILE you created or edited as part of the `@author` tag. Do not write your name.
7. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.
8. To run all the test cases, run `./test.sh`. You can also run the test individually. For instance, run `java Test1 < inputs/Test1.1.in` to execute `Test1` on input `inputs/Test1.1.in`.
9. To check the style of your code, run `./checkstyle.sh`. You can also check the style of individual file with `java -jar checkstyle.jar -c cs2030_checks.xml <FILENAME>`.

IMPORTANT: If the submitted classes or any of the new files you have added cannot be compiled, 0 marks will be given. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

You have been given a class called `CallHistory` that keeps track of the call history of a phone. The class reads a list of phone call information from a file and provides several APIs to print the list of calls, print missed calls, return a call, and calculate the total minutes spent on the phone calls.

The class `CallHistory`, however, is written without following object-oriented principles. You are asked to re-write the class `CallHistory` to adhere to the object-oriented principles you have learned. You may create new classes as needed.

Your revised `CallHistory` must follow the same behavior as the given `CallHistory` (a copy of which can be found in `pristine/CallHistory.java` for your reference). We give an overview of the behavior of `CallHistory` here. For details, please see `CallHistory.java`.

Constructors. An instance of a `CallHistory` can be created using the constructor that takes in a `String` (representing the name of a text file) as an argument. A constructor for `CallHistory` may also take in no argument. In this case, it reads the input from the standard inputs.

```
CallHistory hist1 = new CallHistory(filename); // read from file
CallHistory hist2 = new CallHistory(); // read from standard input
```

Types of Calls. The class can handle three different types of calls.

- Type 0: A call without caller ID, i.e., the phone number of the caller is unknown.
- Type 1: A call with caller ID but cannot be found in the address book, i.e., the phone number is shown but the name of the caller is unknown.
- Type 2: A call with caller ID from a contact found in the address book.

Input File. The file to be loaded into `CallHistory` through the constructor has the following format.

- The first line of the file contains a positive integer n, which is the number of calls.
- The next n lines contain information about the calls. Each of these n lines contains two or more fields, separated by a comma.
  - The first field is always an integer and indicates the types of calls (0, 1, 2)
  - The second field is an integer that indicates the length of the call in minutes. The value of -1 indicates a missed call.
  - The third field applies only if the call comes with caller ID and indicates the phone number of the caller.
  - The fourth field applies only if the caller exists in the addressbook. This field is a string that contains the name of the caller.

You can assume that the given test data follows the input format correctly. A sample input file looks like:

```
6
0,3
1,-1,65554321
0,-1
2,5,95550001,Ahmad
1,10,65554321
2,-1,95551234,Devi
```

Listing Calls. There are two methods provided to list the calls, `printNumbers` and `printAllCalls`.

The method `printNumbers` takes in no arguments and returns nothing. It prints the phone number of each calls enumerated in the same order as they appear in the input files. The string "No Caller ID" is printed if the caller's number is not available.

For example `new CallHistory("Sample.txt").printNumbers()` would print

```
0 No Caller ID
1 65554321
2 No Caller ID
3 95550001
4 65554321
5 95551234
```

The method `printAllCalls` takes in no arguments and returns nothing. It prints the detailed information of each call enumerated in the same order as they appear in the input files, including the phone number, the length of the call (or the string "[MISSED]" if it is a missed call) and the name of the caller (if applicable).

For example `new CallHistory("Sample.txt").printAllCalls()` would print

```
0 No Caller ID | 3 minutes
1 65554321 | [MISSED]
2 No Caller ID | [MISSED]
3 95550001 | 5 minutes | Ahmad
4 65554321 | 10 minutes
5 95551234 | [MISSED] | Devi
```

Calling Back. We can make a call back to one of the listed call records, by calling the method `callback` with the call index as an argument and length of the call in minutes as the argument. If the call was a missed call, it is updated to be a non-missed call. The length of call is added to the total length of the call. For example, after we execute

```
CallHistory hist = new CallHistory("Sample.txt");
hist.callback(1, 24);
hist.callback(3, 7);
hist.printAllCalls();
```

the following will be printed

```
0 No Caller ID | 3 minutes
1 65554321 | 24 minutes
2 No Caller ID | [MISSED]
3 95550001 | 12 minutes | Ahmad
4 65554321 | 10 minutes
5 95551234 | [MISSED] | Devi
```

Note that we cannot return a phone call without caller IDs. If we try

```
hist.callback(0, 24);
```

an error message "Unable to call back: No Caller ID" will be printed.

Print Missed Calls. We can also call the method `printMissedCalls` to print the details of all calls that are missed calls from contacts that can be found in the address book and have not been called back. Executing this snippet

```
CallHistory hist = new CallHistory("Sample.txt");
hist.printMissedCalls();
```

would cause the following to be printed:

```
5 95551234 | [MISSED] | Devi
```

After we execute this:

```
CallHistory hist = new CallHistory("Sample.txt");
hist.callback(5, 3);
hist.printMissedCalls();
```

then nothing would be printed as all missed call from known contact has been returned.

Minutes On Call. The class `CallHistory` provides a method `getMinutesOnCall` that can go through all calls and return the number of minutes spent on calls. For example, executing this snippet

```
CallHistory hist = new CallHistory();
hist.callback(1, 24);
hist.callback(3, 7);
System.out.println(hist.getMinutesOnCall());
```

would print the number 49, which is the sum of all minutes spent on all calls (3 + 24 + 12 + 10).

Your Tasks

Task 1: Rewrite this program using OOP principles

You should read through the file `CallHistory.java` to understand what it is doing. The given implementation applies minimal OO principles, your task in this exam is to rewrite `CallHistory.java`, including adding new classes to apply the OO principles you learned.

To achieve this, create a new class called `Call` to encapsulate the relevant attributes and methods. Create subclasses of `Call` as necessary. Use polymorphism to simplify the code in `CallHistory` and make your code extensible to possible new call types in the future. Make sure all OO principles, including LSP, tell-don't-ask, information hiding, are adhered to.

Solution: Call.java

```
abstract class Call {
  private int length;
  private boolean isMissed;

  public Call(int length) {
    if (length == -1) {
      this.isMissed = true;
    } else {
      this.length = length;
      this.isMissed = false;
    }
  }

  public int getCallLength() {
    return this.length;
  }

  public void callback(int length) throws IllegalCallException {
    this.length += length;
    this.isMissed = false;
  }

  public abstract boolean isKnown();

  public boolean isMissed() {
```

```
    return this.isMissed;
  }

  public String toPrettyString() {
    if (this.isMissed) {
      return "[MISSED]";
    } else {
      return this.length + " minutes";
    }
  }
}
```

NumberedCall.java

```
abstract class NumberedCall extends Call {
  private int number;

  public NumberedCall(int length, int number) {
    super(length);
    this.number = number;
  }

  @Override
  public String toString() {
    return "" + number;
  }

  @Override
  public String toPrettyString() {
    return number + " | " + super.toPrettyString();
  }
}
```

ContactCall.java

```
class ContactCall extends NumberedCall {
  private String who;

  public ContactCall(int length, int number, String who) {
    super(length, number);
    this.who = who;
  }

  @Override
  public boolean isKnown() {
    return true;
  }

  @Override
  public String toPrettyString() {
    return super.toPrettyString() + " | " + this.who;
  }
}
```

UnknownCall.java

```
class UnknownCall extends NumberedCall {
  public UnknownCall(int length, int number) {
    super(length, number);
  }

  @Override
  public boolean isKnown() {
     return false;
  }
}
```

NoCallerIdCall.java

```
class NoCallerIdCall extends Call {

  public NoCallerIdCall(int length) {
    super(length);
  }

  @Override
  public String toString() {
    return "No Caller ID";
  }

  @Override
  public boolean isKnown() {
    return false;
  }

  @Override
  public String toPrettyString() {
    return "No Caller ID" + " | " + super.toPrettyString();
  }

  @Override
  public void callback(int length) throws IllegalCallException {
    throw new IllegalCallException(this);
  }
}
```

Task 2: Implement exception handling
Create a new exception class called `IllegalCallException` that is thrown when `callback` is invoked on a a call with no caller ID. The exception should be initialized with the error message "Unable to call back: No Caller ID".

This exception should be caught and handled in the method `callback` of `CallHistory`.

Solution: IllegalCallException.java

```
class IllegalCallException extends Exception {
  private Call e;

  public IllegalCallException(Call e) {
```

```
    this.e = e;
  }

  @Override
  public String toString() {
    return "Unable to call back: " + e;
  }
}
```

CallHistory.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class CallHistory {
  private Array<Call> array;

  private static final int NO_CALLER_ID = 0;
  private static final int UNKNOWN_CALLER = 1;
  private static final int CONTACT = 2;

  public CallHistory() {
    this(new Scanner(System.in));
  }

  public CallHistory(String filename) throws FileNotFoundException {
    this(new Scanner(new File(filename)));
  }

  private CallHistory(Scanner sc) {
    loadCalls(sc);
  }

  private void loadCalls(Scanner sc) {
    int numOfCalls = Integer.parseInt(sc.nextLine().trim());
    array = new Array<Call>(numOfCalls);
    int i = 0;
    while (sc.hasNext()) {
      String text = sc.nextLine().trim();
      String arguments[] = text.split(",");
      array.set(i, createCall(arguments));
      i = i + 1;
    }
  }

  private Call createCall(String[] args) {
    int type = Integer.parseInt(args[0]);
    int length = Integer.parseInt(args[1]);
    Call t = null;
    if (type == CallHistory.NO_CALLER_ID) {
      t = new NoCallerIdCall(length);

    } else if (type == CallHistory.UNKNOWN_CALLER) {
      int number = Integer.parseInt(args[2]);
```

```
      t = new UnknownCall(length, number);

    } else if (type == CallHistory.CONTACT) {
      int number = Integer.parseInt(args[2]);
      String name = args[3];
      t = new ContactCall(length, number, name);
    }
    return t;
  }

  @Override
  public String toString() {
    String s = "";
    int i = 0;
    for (i = 0; i < array.length(); i++) {
      s = s + "\n" + array.get(i);
    }
    return s;
  }

  public int getMinutesOnCall() {
    int sum = 0;
    for (int i = 0; i < array.length(); i++) {
      Call call = array.get(i);
      sum += call.getCallLength();
    }
    return sum;
  }

  public void printNumbers() {
    for (int i = 0; i < array.length(); i++) {
      Call call = array.get(i);
      System.out.println(i + " " + call);
    }
  }

  public void printAllCalls() {
    for (int i = 0; i < array.length(); i++) {
      Call call = array.get(i);
      System.out.println(i + " " + call.toPrettyString());
    }
  }

  public void printMissedCalls() {
    for (int i = 0; i < array.length(); i++) {
      Call call = array.get(i);
      if (call.isMissed() && call.isKnown()) {
        System.out.println(i + " " + call.toPrettyString());
      }
    }
  }

  public void callback(int i, int length) {
    try {
      array.get(i).callback(length);
```

```
    } catch (IllegalCallException e) {
      System.out.println(e);
    }
  }
}
```

Reminder: all checked exceptions are a subclass of `java.lang.Exception`. The class `Exception` has the following constructor:

```
Exception(String msg)
```

that constructs a new exception with the specified detail message `msg`. The message can be retrieved by the `getMessage()` method, which returns the message as a `String`.

END OF PAPER