CS1002 W07 Practical Report

Student ID: 180004823 Tutor: Adriana Wilde

November 2, 2018

Overview

For this task I was asked to make two classes - one for people and the other for games. These would have suitable variables and methods to perform a number of tasks. As described in the specification the program should be able to:

- Calculate the current age of a person.
- Print out the details of a person in a readable format.
- Pay a certain amount into a player's balance.
- Allow a person to join a game if certain conditions are met.
- Allow a person to leave a game.
- Print out the details of a game.
- Keep track of total amount accumulated by a player in fees.

This is then implemented with classes to represent games and people with suitable attributes to support these operations. Instances of the classes can be created and tested within the main method of a class called W07Practical.

Design

When designing my solution I first had to establish what classes were needed and the attributes these classes would need. I decided outside of the class that the program was being executed from it would be sensible to have two additional classes - Person and Game. Person would store details about players within the program, they would have the attributes of id (which would be stored as an integer), name (stored as a string), date of birth (stored as the Java Date object), amount owed (stored as an integer representing the number of pence in the person's balance), and the total amount accumulated (stored as an int).

The Game class must have the attributes of id (stored as an int), name (stored as a string), minimum age (stored as an int), and fee (stored as an int in pence). The Game must also have a data structure to store the players currently playing the game, I decided to use an ArrayList as this structure is dynamic and thus can be expanded if more than two players are needed in future. There must also be a variable to store the maximum number of players which can be changed in future to allow for expansion.

Next I wanted to design the methods for each class. The first thing that needed to be done was to calculate the age of a person. Because there is no native way to find the number of years between two dates I would get the millisecond time of each date subtract and then do a calculation to convert to years. Because of the high amount of accuracy needed I would have to use a data type of large accuracy, therefore I chose to use doubles for the calculations.

To output the details of a person I planned to override the toString method. This means to print out the details of a person I would just have to call System.out.println() with that particular person's corresponding object as the argument. The toString() method would start with an empty string and append each attribute of the person as an individual line so that the information is clear. The final line would be a row of symbols to separate between outputs if lots of different objects are being printed to the console.

Next I would need a method to pay an amount into a person's balance. First I would need a conditional to check if the amount being paid is less than or equal to the balance of their account. If this is true then the amount paid can be subtracted from the balance. If not then an error message needs to be outputted.

Next I had to write a method within the Game class to allow players to join the game. The conditions for this would be:

- Person's age is greater than the minimum age.
- The number of player's playing the game was less than the maximum number of players.
- The person is not already playing the game.

Once it was established that these conditions were met than the person could be added to that game's list of currentPlayers and the persons balance can be increased by the games fee. If any of these conditions were not met then an appropriate error message should be provided. An appropriate logical structure for this would be a nested if statement so that if any particular condition is not met than the error message can be customised to inform the user.

As well as allowing people to join the game, the class Game must have a function to allow people to leave the game. The only conditional needed here is to check if the person that the program is trying to remove is actually playing the game. If not then an appropriate error message should be provided. If the person is playing the game then all the program has to do is remove them from the currentPlayers list.

As with the Person class I planned to override the toString method within the Game class. The method was pretty much the same as the equivalent method within the Person class except that at the end I printed the array of people's names that were stored within the currentPlayers list.

The last requirement of the specification was to keep track of the total amount accumulated by a player in fees. I planned to do this by having an attribute within Person separate to the balance that was increased whenever the player entered a game however was not deducted from when the person paid into their balance.

All the attributes of Person and Game were to be kept private with public methods to access and edit them. For example within the method in Game to add a player the method increaseBalance() would be called on that player which would both increase the balance and accumulated total attributes. This improves my control over the code and understanding of what is happening as well as making the code more easy to maintain when I add to it in the extension.

To make the Ids stored within all the objects unique each class must have a static attribute called nextID and a static method called getNextID. This is used so when a new object is made its id attribute is assigned to the nextID attribute and the nextID attribute is increased. This nextID attribute is static and thus is not specific to the object it is being called from rather is generalised to the class as a whole.

The getters and setters must ensure that incorrect information is not being added to the class. Below are examples of the sanitisation of inputs that needs to happen:

- setId in both classes needs to always set the id to the next id to avoid id clashes by using the static method described above.
- setDateOfBirth in Person must ensure entered date of birth is after the current date. If date is after set date to current date and return error message.
- setAmountOwed in Person must be a number greater than or equal to 0. If not then set amount to 0 and return error message.
- setMinimumAge in Game must ensure input is greater than or equal to 0. If not then set to 0 and return error message.
- setFee in Game must ensure input is also greater than or equal to 0. If not then set to 0 and return error message.
- setMaximumPlayers must once again check if input is greater than or equal to 1 (a 0 player game is pointless), if not set to 2 and return error message.

Testing

Within this section I will use the main class to create objects that will be used to test whether the classes Game and Person are functioning as intended. First I will test the creation and printing of these objects by testing a variety of inputs and confirming that the output is as expected.

Testing of the Creation and Printing of Person Objects

To create a person the following code is used within the main method:

```java
    SimpleDateFormat simpleDateFormat = new
        SimpleDateFormat("dd-MM-yyyy");

      try {
          Date date = simpleDateFormat.parse("04-06-1999");
          Person dave = new Person("Dave", date, 10);
          System.out.println(dave);
      }
      catch (ParseException e) {
          System.out.println("Error: invalid date entered");
      }
```

This code provides the following output:

```
PLAYER INFORMATION
id : 1
name : Dave
age : 19
current balance owed : 10
=====================
```

For the sake of ease of reading I will not be providing this full print out for every test rather I will be using a table to show the output from the id, name, age, and current balance owed attributes.

Date In	**Name In**	**Balance In**	**Id Out**	**Name Out**	**Age Out**	**Balance Out**	**Result**
04-06-1999	Dave	10	1	Dave	19	10	Pass
02-03-2000	Bob	100	2	Bob	18	100	Pass
30-11-2019	Dick	-1000	3	Dick	0	0	Pass
2000-10-20	Dom	30	N/A	N/A	N/A	N/A	Pass
04-05-1801	Mildred	1	4	Mildred	217	1	Pass
10-30-2018	Leo	0	5	Leo	0	0	Pass

To thoroughly test the program I provided it with a range of inputs. The first two were very generic just to test that some output was being provided. Both provided completely correct outputs as shown in the table. For the third row's inputs I provided both an invalid date input and balance input. The following error messages were printed to the console:

```
Error: that date of birth is in the future, date of birth set to current date
Error: that amount is less than 0, amount owed set to 0
```

Both inputs were sanitised and set to the defaults I assigned in this scenario. For the fourth test I made the date invalid. This threw an error in the code providing the error message "Error: invalid date entered" because of this the object wasn't even created. Hence the N/As. For the Mildred object I tested an extremely old date to see if the program handled it correctly, which it did. Finally for the Leo object I tested the current date (10/30/2018) to check if the program would recognise it as a valid date and the age as 1 - which it did.

Testing of the Creation and Printing of Game Objects
To create and print a Game object I use the following code in the main method:

```
    Game portal = new Game("Portal", 12, 50);
    System.out.println(portal);
```

This gave the output following output:

```
GAME INFORMATION
id : 1
```

```
name : Portal
minimum age : 12
fee : 50
number of players in game : 0
players :
=====================
```

As with the testing of the creation and printing of the Person class for the Game class I will not be providing this printout for every example as it would make the testing more difficult to read instead I will be using a table to show the outputs for id, name, age, and fee. I will not yet test the number of players field as this will be another area of the program to isolate and test.

Name In	**Age In**	**Fee In**	**Id Out**	**Name Out**	**Age Out**	**Fee Out**	**Result**
Portal	12	50	1	Portal	12	50	Pass
Mario	18	100	2	Red Dead Redemption	18	100	Pass
Civilization	15	75	3	Civilization	15	75	Pass
Pacman	0	50	4	Pacman	0	50	Pass
Dig Dug	-5	-30	5	Dig Dug	0	0	Pass
Donkey Kong	10000	10000	6	Donkey Kong	10000	10000	Pass

As seen from the table the program can create and print the Game objects as specified by the input. If the user inputs something invalid such as the negative values within the Dig Dug object then the program sanitizes this input and sets the values of these attributes to the default of 0. Because of the invalid inputs the following error messages are provide:

```
Error: minimum age cannot be less than 0, minimum age has been set to 0
Error: fee cannot be less than 0, fee has been set to 0
```

Testing Adding Money to Person's Account

The next thing that needed to be performed by my program was the ability to pay money into a player's balance below is the syntax for how my program performs this functionality:

```
    person.payMoney(100);
```

I have already shown that the balance of a player can only be set to a positive value so for the sake of testing this part of the program I am going to create a player with a balance of 100 and call the method payMoney on the object with a range of input parameters. I will then print the output as shown to work before.

Amount Added	**Result**	**Comments**
60	40	Passed
70	30	Passed
80	20	Passed
90	10	Passed
100	0	Passed
110	0	Passed - Correct Error Message Printed

For the last test the error message provided printed: ”Error: cannot pay more than is owed”.

Testing Adding People to a Game

Because of the nature of these tests it is not appropriate instead I shall test each of the conditions that I had previously mentioned need to be met for a player to be added, namely:

- Person is older than minimum age for the game.
- Game is not already full.
- Person is not already playing the game.

To test the first condition let’s test whether a person younger than 18 and older than 18 can play an 18+ game. First I will create a game with an age limit of 18 and test adding an underage person to it:

```
    Game silentHill = new Game("Silent Hill", 18, 50);
      try {
          Person underage = new Person("Jack",
              simpleDateFormat.parse("03-04-2004"), 100)
          Person overage = new Person("Harry",
              simpleDateFormat"02-03-2000", 100);

          silentHill.addPlayer(underage);
      }
      catch (ParseException e) {
          System.out.println("Error : invalid date");
      }
```

The following output is provided in the console:
```
Error: player too young to play this game
```

I am going to repeat the test instead passing the overage person into the addPlayer function, this time I was provided with no error and upon printing the silentHill Game object I get the following output:

```
GAME INFORMATION
id : 1
name : Silent Hill
minimum age : 18
fee : 50
number of players in game : 1
players : Harry
=====================
```

As you can see the number of players has increased to one and the name of the overage player has been outputted within the players output. I have now confirmed that the program will not add someone if they are underage so the next test is to see if the program prevents you from adding too many players to a game. To do this I will create a game and keep adding players and see if the program allows you to add more than two players to one game. Below is the code used to test this:

```
    SimpleDateFormat simpleDateFormat = new
        SimpleDateFormat("dd-MM-yyyy");
    Game mario = new Game("Super Mario", 0, 50);
      try {
          Person jack = new Person("Jack",
              simpleDateFormat.parse("03-04-2004"), 100);
          Person harry = new Person("Harry",
              simpleDateFormat.parse("02-03-2000"), 100);

          mario.addPlayer(jack);
          mario.addPlayer(harry);

          System.out.println(mario);
      }
      catch (ParseException e) {
          System.out.println("Error : invalid date");
      }
```

Provided below is the output of the code:

```
GAME INFORMATION
id : 1
name : Super Mario
minimum age : 0
fee : 50
number of players in game : 2
players : Jack Harry
======================
```

As you can see from the output the number of players has increased to 2 and the names of the two players are correctly being displayed. Next I need to check what the program does if you try and add more than two players. Below is the code used to test this:

```
    SimpleDateFormat simpleDateFormat = new
        SimpleDateFormat("dd-MM-yyyy");

      Game mario = new Game("Super Mario", 0, 50);
      try {
          Person jack = new Person("Jack",
              simpleDateFormat.parse("03-04-2004"), 100);
          Person bill = new Person("Harry",
              simpleDateFormat.parse("02-03-2000"), 100);
          Person janet = new Person("Janet",
              simpleDateFormat.parse("03-04-1998"), 60);

          mario.addPlayer(jack);
          mario.addPlayer(bill);
          mario.addPlayer(janet);

          System.out.println(mario);
      }
```

```
        catch (ParseException e) {
            System.out.println("Error : invalid date");
        }
```

Provided below is the output of the code:

```
Error: only 2 can play this game at any one time
GAME INFORMATION
id : 1
name : Super Mario
minimum age : 0
fee : 50
number of players in game : 2
players : Jack Harry
=====================
```

The final condition I need to test when adding a new player is whether the program will stop the same person being added to the game twice. To test this I ran the following code:

```
      SimpleDateFormat simpleDateFormat = new
          SimpleDateFormat("dd-MM-yyyy");

        Game mario = new Game("Super Mario", 0, 50);
        try {
            Person jack = new Person("Jack",
                simpleDateFormat.parse("03-04-2004"), 100);

            mario.addPlayer(jack);
            mario.addPlayer(jack);

            System.out.println(mario);
        }
        catch (ParseException e) {
            System.out.println("Error : invalid date");
        }
```

Provided below is the output of the code:

```
Error: player number 1 is already in the game
GAME INFORMATION
id : 1
name : Super Mario
minimum age : 0
fee : 50
number of players in game : 1
players : Jack
=====================
```

As you can see from the output an error message is displayed informing you that you cannot add the same player twice. The player Jack is not added to the players list as shown from the GAME INFORMATION output. This means this test has been passed.

Testing Removing a Person From a Game

When trying to remove a person from a game the only condition that needs to be met to complete the process is for that person to already be in that game in the first place. Therefore I need only test that the program can deal with this conditions. First I will show that you are able to remove someone who is in the game. Below is the code used to test this:

```
     SimpleDateFormat simpleDateFormat = new
         SimpleDateFormat("dd-MM-yyyy");

       Game mario = new Game("Super Mario", 0, 50);
       try {
           Person jack = new Person("Jack",
               simpleDateFormat.parse("03-04-2004"), 100);

           mario.addPlayer(jack);

           System.out.println(mario);

           mario.removePlayer(jack);

           System.out.println(mario);
       }
       catch (ParseException e) {
           System.out.println("Error : invalid date");
       }
```

Shown below is the output of this code:

```
GAME INFORMATION
id : 1
name : Super Mario
minimum age : 0
fee : 50
number of players in game : 1
players : Jack
======================
GAME INFORMATION
id : 1
name : Super Mario
minimum age : 0
fee : 50
number of players in game : 0
players :
======================
```

The first printed section game information is after adding the player, as you can see the number of players is 1 and that player is Jack (who's player id is 1). The second printed section is after jack is removed from the game. The player number is now 0 as it should be. Next I need to test what happens if you try and remove a player who is not in the game. Below is the code used to check this:

```
     SimpleDateFormat simpleDateFormat = new
         SimpleDateFormat("dd-MM-yyyy");

       Game mario = new Game("Super Mario", 0, 50);
       try {
           Person jack = new Person("Jack",
               simpleDateFormat.parse("03-04-2004"), 100);

           mario.removePlayer(jack);
       }
       catch (ParseException e) {
           System.out.println("Error : invalid date");
       }
```

Below is the output given by the execution of the code:

```
Error: player number 1 is not in this game
```

This is the correct output as the player id of Jack is 1 and the program has recognised that he is not in the game.

Testing Keeping Track of Accumulated Fees The final part of the specification I need to prove works is that the program keeps track of accumulated fees. This is done within the same function that increases the balance of the player as a variable is increased whenever a player joins the game however this variable is not changed when money is paid in. Below is the code I used to test if this functionality works.

```
     SimpleDateFormat simpleDateFormat = new
         SimpleDateFormat("dd-MM-yyyy");

       Game mario = new Game("Super Mario", 0, 50);
       try {
           Person jack = new Person("Jack",
               simpleDateFormat.parse("03-04-2004"), 100);

        System.out.println(jack.getTotalAccumulated());
           mario.addPlayer(jack);
           System.out.println(jack.getTotalAccumulated());
           jack.payMoney(100);
           System.out.println(jack.getTotalAccumulated());
       }
       catch (ParseException e) {
           System.out.println("Error : invalid date");
       }
```

Below is the output of the code:

```
0
50
50
```

From the output you can tell that before Jack joined any games his total accumulated amount was 0 then after joining a game this was increased by the game's fee of 50. When the balance was paid the total accumulated wasn't altered. This shows that the functionality works.

Question

The way I programmed my solution planned for the eventuality that more players could play the game as I used a list of people within the Game class to store the players currently playing and used another integer variable to declare the maximum size of this list. However if, instead, I used two attributes for each player to store the people currently playing then it would become impractical to expand this to larger game sizes.

Evaluation

As shown by the test section, my program fulfills all the requirements of the specification. You can create Game and Person objects, the details of which can be printed in an easy to read format to the console. Each person has a balance which keeps track of what they need to pay for their games. Two people can be added to each game and people can also leave games. The program is also able to keep track of the total amount accumulated in fees by any player.

Conclusion

During the development of this program I was able to demonstrate lots of important aspects of object oriented programming. The first was the creation of objects from a class template with a constructor. Using a constructor reduced the amount of code I had to write as I didn't have to individually set the attributes of the objects. I also made use of encapsulation by having the attributes of both the Game and Person classes private. I could access and change the variables via the use of getters and setters which meant I could ensure that the values of those attributes were valid. For example I made the setters for a few attributes such that they cannot be negative.

I also separated lots of the code into smaller, simpler methods. This made the testing of the code much more efficient as I could isolate problems more easily and each smaller part of the code only required a small amount of testing. I found certain parts of the code quite hard to provide tests for as I cannot concretely prove that there are no possible bugs I just had to be very careful to try and prepare for every situation through the use of if statements and try catch statements. The use of try catch to prevent invalid user input was handy.

In the extension I plan to add a few things to make the program more complex. I will allow the Game class to have any given number of players up to a maximum decided by the user. I will do the suggested tasks in the extension such as introduce more conditions for joining a game and having a bank account system.

Extension

Design

The first part of the extension I wanted to do were the additional conditions for a player being added to a game. The first of these conditions was to only allow the player to play games 100 times. I planned to do this by having an attribute within the Person class that stores the number of games they had joined and a condition within the Game class to check if this amount is less than or equal to 100. The second of these conditions was to ensure that the player doesn't owe more than a certain amount. The limit by default should be 1000 but this can be changed via the use of getters and setters with the setter only allowing positive values.

The next part of the extension I wanted to do was allowing any number of players. I did this by taking the maximum players attribute as a parameter within the constructor. When you are adding a player the currentPlayers list's size is compared to the maximum player number to check if there is space in the game. This is only possible as when designing the original solution I had the foresight to store the players in a game in a list rather than in individual variables.

Finally I wanted to implement "bank accounts", these would be objects assigned to a person from which they can withdraw money to pay into the game. Each person has a bank account which can be used to pay money. The transaction will only be completed if the person has enough money in the bank account. The bank account class should have the attributes account number (an int), date created (java Date object), and the balance (an int stored in pennies).

Testing

Testing the Additional Conditions When Adding Players

Firstly I'm going to test the 100 games rule. I'm going to use a for loop to add a person to a game and then remove them over and over and test if the program doesn't allow the persons to play more than 100 times. Below is the code used to test this:

```
    SimpleDateFormat simpleDateFormat = new
        SimpleDateFormat("dd-MM-yyyy");

      try {
          Game zelda = new Game("Legend of Zelda", 5, 1, 1);
          BankAccount bankAccount = new
              BankAccount(simpleDateFormat.parse("03-03-2015"), 100);
          Person dave = new Person("Dave",
              simpleDateFormat.parse("02-04-1999"), 0, bankAccount);

          for(int i = 0; i < 100; i++) {
              zelda.addPlayer(dave);
              zelda.removePlayer(dave);
          }
      } catch (ParseException e) {
          e.printStackTrace();
      }
```

As you can see the person is being added and removed 100 times. No error messages are provided as expected. Instead I increased the limit in the for loop so that the player is being added and removed 101 times. This time the following error message is provided:

```
Error: person has joined 100 games, limit has been reached
Error: player number 1 is not in this game
```

From this we can see that the program has correctly identified the player has exceeded the game limit. The second error is because the program is attempting to remove the player from the game when they have not been added (correctly so). This means the program has performed this part of the functionality correctly.

The next condition I need to check is the balance limit. The way I shall do this is by testing whether the game will allow a player who is above or below the limit and see what the program outputs. As shown in the previous test, a person who is below the limit can join. I shall use the code below to test for a player who is above the limit:

```
      SimpleDateFormat simpleDateFormat = new
          SimpleDateFormat("dd-MM-yyyy");

        try {
            Game zelda = new Game("Legend of Zelda", 5, 1, 1);
            zelda.setMaxOwed(100);
            BankAccount bankAccount = new
                BankAccount(simpleDateFormat.parse("03-03-2015"), 100);
            Person dave = new Person("Dave",
                simpleDateFormat.parse("02-04-1999"), 200, bankAccount);

            zelda.addPlayer(dave);

        } catch (ParseException e) {
            e.printStackTrace();
        }
```

Below is the output for the code:

```
Error: person owes more than is allowed for this game
```

As you can see the program has correctly recognised that the person has exceeded the limit for the amount they owe to play the game so this part of the program's functionality works.

Testing Having Different Amounts of People Within a Game

Because I cannot feasibly test every number of players I will be focusing on 1 and 3, I have shown that 2 players work and so it would make sense to test a case that is above and below this number. Firstly this is the code used to test one player:

```
     SimpleDateFormat simpleDateFormat = new
         SimpleDateFormat("dd-MM-yyyy");

       try {
           Game zelda = new Game("Legend of Zelda", 5, 1, 1);
           zelda.setMaxOwed(100);
           BankAccount bankAccount = new
               BankAccount(simpleDateFormat.parse("03-03-2015"), 100);
           Person dave = new Person("Dave",
               simpleDateFormat.parse("02-04-1999"), 0, bankAccount);
           Person mike = new Person("Mike",
               simpleDateFormat.parse("03-06-1999"), 0, bankAccount);

           zelda.addPlayer(dave);
           System.out.println(zelda);

           zelda.addPlayer(mike);
           System.out.println(zelda);

       } catch (ParseException e) {
           e.printStackTrace();
       }
```

Below is the output of this code:

```
GAME INFORMATION
id : 1
name : Legend of Zelda
minimum age : 5
fee : 1
number of players in game : 1
players : Dave
======================
Error: only 1 can play this game at any one time
GAME INFORMATION
id : 1
name : Legend of Zelda
minimum age : 5
fee : 1
number of players in game : 1
players : Dave
======================
```

As you can see the first person, Dave, is successfully added as displayed by the printed output. When the next person Mike is attempted to be added the program recognises that the game is full and outputs an error message. You can tell that the program hasn't added Mike as he is not outputted in the players list in the second game information output.

Next I have to test a game for three players, below is the code used to test this:

```
    SimpleDateFormat simpleDateFormat = new
        SimpleDateFormat("dd-MM-yyyy");

      try {
        Game marioKart = new Game("Mario Kart", 5, 1, 3);
        marioKart.setMaxOwed(100);
        BankAccount bankAccount = new
            BankAccount(simpleDateFormat.parse("03-03-2015"), 100);
        Person dave = new Person("Dave",
            simpleDateFormat.parse("02-04-1999"), 0, bankAccount);
        Person mike = new Person("Mike",
            simpleDateFormat.parse("03-06-1999"), 0, bankAccount);
        Person janet = new Person("Janet",
            simpleDateFormat.parse("21-10-1921"), 0, bankAccount);
        Person jill = new Person("Jill",
            simpleDateFormat.parse("04-11-1978"), 0, bankAccount);

        marioKart.addPlayer(dave);
        marioKart.addPlayer(mike);
        marioKart.addPlayer(janet);
        System.out.println(marioKart);

        marioKart.addPlayer(jill);
        System.out.println(marioKart);

      } catch (ParseException e) {
        e.printStackTrace();
      }
```

Below is the output of the code:

```
GAME INFORMATION
id : 1
name : Mario Kart
minimum age : 5
fee : 1
number of players in game : 3
players : Dave Mike Janet
======================
Error: only 3 can play this game at any one time
GAME INFORMATION
id : 1
name : Mario Kart
minimum age : 5
fee : 1
number of players in game : 3
players : Dave Mike Janet
```

```
=======================
```

The first game information section is outputted after the first three people have been added, as you can see these people have successfully been added to the list of players. When the program attempts to add another person an error message is provided and the person is not added as shown by the players list being the same. From these tests I have deemed the program to work with games of different sizes.

Testing the Bank Account System

The last extension feature I need to test is the bank account objects. These are objects that have a balance that can be withdrawn from to pay into a Person's owed game balance. To test this I will create a bank with a certain amount of balance and then withdraw money until the bank account is empty. Here is the code used to test this:

```java
    SimpleDateFormat simpleDateFormat = new
        SimpleDateFormat("dd-MM-yyyy");

      try {
          Game marioKart = new Game("Mario Kart", 5, 50, 3);
          marioKart.setMaxOwed(1000);
          BankAccount bankAccount = new
              BankAccount(simpleDateFormat.parse("03-03-2015"), 150);
          Person dave = new Person("Dave",
              simpleDateFormat.parse("02-04-1999"), 0, bankAccount);

          // have dave play mario kart once
          marioKart.addPlayer(dave);
          marioKart.removePlayer(dave);

          // pay the balance and then print out what's remaining in
              bank account
          System.out.println("Balance to Pay: " + dave.getAmountOwed());
          dave.payMoney(50);
          System.out.println("Money Left in Bank Account: " +
              bankAccount.getBalance());

          // have dave play mario kart twice more
          marioKart.addPlayer(dave);
          marioKart.removePlayer(dave);
          marioKart.addPlayer(dave);
          marioKart.removePlayer(dave);

          // pay the balance again and print out what's remaining in
              bank account
          System.out.println("Balance to Pay: " + dave.getAmountOwed());
          dave.payMoney(100);
          System.out.println("Money Left in Bank Account: "+
              bankAccount.getBalance());

          // have dave play the game one more time
          marioKart.addPlayer(dave);
```

```
            marioKart.removePlayer(dave);

            // attempt to pay the balance
            System.out.println("Balance to Pay: " + dave.getAmountOwed());
            dave.payMoney(50);
            System.out.println("Money Left in Bank Account " +
                bankAccount.getBalance());

        } catch (ParseException e) {
            e.printStackTrace();
        }
```

Below is the output from running this code:

```
Balance to Pay: 50
Money Left in Bank Account: 100
Balance to Pay: 100
Money Left in Bank Account: 0
Balance to Pay: 50
Error: not enough money in bank account
Money Left in Bank Account 0
```

As you can see the bank account empties as money is paid into the games balance to pay. When there is no money left in the account and the program attempts to do a withdrawal the program informs the user of the error and doesn't deduct from the bank balance. From this test I can safely say the bank account feature works.

Evaluation

When I designed the program I laid out the three features I wanted to add. Firstly I wanted to add two conditions when adding players to a game, first checking they had not been in too many games and secondly checking that they do not owe more than certain amount. As shown by the tests this feature works properly. The second thing I wanted to add was the ability to have a different number of people in a game. This is because many games vary in how many people play the game. As shown in the testing section this works as expected. The finally thing I wanted to have was a bank account system such that money can be withdrawn to pay into an account. This was tested in the testing section and works properly.

Conclusion

In this extension I was able to make use of the dynamic data structure ArrayList to vary the number of players in a game. Dynamic means that the size of the structure can change during after the structure is initialised and is helpful as I do not know the size of the game before the constructor is called. I was able to further use Java's object oriented nature when programming the BankAccount class as instances of this class can be stored within the Person class which can then be used to pay money.

I found it quite difficult to provide rigorous testing as I cannot simply compare an outputted value to an expected value as I might be able to do with simpler programs. I also can't prove with absolute certainty that there are no bugs in the code. If given more time I would create a GUI to give the program a user friendly interface. I would also provide a log-in system so a user can track how much they owe.