Assignment5 MONOPOLY

General Designer: ZHU Yueming

Official Judger Designer: YIN Chaoyue

Tester: FENG Haibo

What to Submit:

Sutmit at least those five java files to JCoder.

LandColor.java

Player.java

House.java

Land.java

GameSystem.java

Introduce

This is a Monopoly game with 2-4 **players** and an 8-30 **lands**.

- Each **land** has its own color.
- **Players** can take 1-6 steps in each turn. After player reaches the last **land**, they will return to the original **land** to continue walking.
- The initial money of each player is 100.
- **Players** can build or not build a **house** on an empty **land**, but if a player occupies a land of a color, other players are not allowed to build a house on the other land with the same color.
- If the player reaches a location where there is an opponent's house, they need to pay 50% of the house price to house owner.
- If the player doesn't have enough money to pay the rent, then the player is failed.

Classes

Class 1: LandColor

It is an enum class, which represents the color of Land. We provide 8 colors below:

```java
public enum LandColor {
    RED, ORANGE, YELLOW, GREEN, BLUE, PURPLE, WHITE, BLACK;
}
```

> For this class, do not modify or remove any enum instance that have been already defined, however you can add fields and methods if you need.

Class 2: Player

Fields:

- **id**: Represents the id of player. It is an integer value, and in our test cases different players do not have a same id.

  ```
  private int id;
  ```

- **money**: Represents player's money. The initial value of money is **100**.

  ```
  private double money;
  ```

- **isActive**: Represent whether the player is failed the game. The initial value of isActive is **true**. If the player cannot have enougth money to pay the rent, the isActive should be false.

  ```
  private boolean isActive;
  ```

- **location**: Represent the location of the game. It is an integer value, and the initial value of location is **0**.

  ```
  private int location;
  ```

Methods:

- **Constructor**: Only has one parameter id, but you need to initialize other private fields in constructor.

  ```
  public Player(int id)
  ```

- **getter and setter**: Create getter and setter methods for all private fields.
- **toString()**: The string format is according to isActive value.

  ```
  public String toString()
  ```

  - If the player is active, the String format is: `Player [id]: at [location], has [money]`, for example: (money should be rounded to 1decimal place)

    ```
    Player 1: at 0, has 25.0
    Player 2: at 1, has 42.5
    ```

  - If the player isn't active, the String format is: `Player [id]: Failed`, for example:

```
Player 1: Failed
Player 2: Failed
```

- **payRent()**: According to the parameter `housePrice`, and return a double value which represents how much money actually paid.

  ```
  public double payRent(int housePrice)
  ```

  - The transaction amount is 0.5 times of housePrice
  - When money is enough, the money should minus transaction amount, and then return the transaction amount.
  - When money is not enough, which means the player is failed and you need to change the isActive to false, and then return the left money. **Hint: if a player is not active, we would not test how many money the player left.**

  For example:

  ```
  Player p1 = new Player(1);
  System.out.println(p1.payRent(100));
  System.out.println(p1);
  System.out.println(p1.payRent(101));
  System.out.println(p1);
  ```

  Result:

  ```
  50.0
  Player 1: at 0, has 50.0
  50.0
  Player 1: Failed
  ```

- **collectRent()**: According to the parameter `rentPrice`, and then add the rentPrice into money.

  ```
  public void collectRent(double rentPrice)
  ```

- **buildHouse**: This method will be introduced later on.

  ```
  public boolean buildHouse(Land land, int housePrice)
  ```

Class 3: House

> For this class, do not modify or remove any fields and methods that have been already defined, however you can add other fields or methods if you need.

Fields:

- **player**: Record which player create the house.

  ```
  private Player player;
  ```

- **housePrice**: How much the price is.

  ```
  private int housePrice;
  ```

Methods

Constructor: Has two parameter constructor as below:

```
 public House(Player player, int housePrice) {
        this.player = player;
        this.housePrice = housePrice;
    }
```

- **getter and setter**: Create getter and setter methods for all private fields.
- **toString()** : The toString() method can be designed as below:

```
@Override
public String toString() {
    return String.format("[H P%d: %d]", player.getId(), housePrice);
}
```

Class 4: Land

> For this class, do not modify or remove any fields and methods that have been already defined, however you can add other fields or methods if you need.

Fields:

- **house**: The house built in the land. The initial value of house is null. There can be at most one house on a land.

  ```
  private House house;
  ```

- **color**: The color of this land, which type is LandColor.

```
private LandColor color;
```

Methods:

- **Constructor**: Has one parameter constructor as below:

```
 public Land(LandColor color) {
        this.color = color;
    }
```

- **getter and setter**: Create getter and setter methods for all private fields.
- **toString()** : The toString() method can be designed as below:

```
public String toString() {
        return String.format("Land %s: %s", this.color, house != null ?
house.toString() : "");
    }
```

Class 2: Player (BuildHouse Method)

- **buildHouse**:

```
public boolean buildHouse(Land land, int housePrice)
```

The return value would be false and the method would do nothing when one of following events happen:

- The parameter `land` has a house.
- The parameter `housePrice` is zero.
- There is not enough `money` to pay the `housePrice`.

If the player build a house successfully:

- The money should minus housePrice.
- Create a new house in this land.
- return a true value.

> Hint: We would not check whether the color of land is occupied by other player in this method.

For example:

```
Player p1 = new Player(1);
Player p2 = new Player(2);

Land l1 = new Land(LandColor.RED);
Land l2 = new Land(LandColor.BLUE);
Land l3 = new Land(LandColor.BLACK);
```

```
System.out.println(p1.buildHouse(l1, 55));//true
System.out.println(p1.buildHouse(l2, 100));//false
System.out.println(p2.buildHouse(l1, 50));//false
System.out.println(p2.buildHouse(l2, 0));//false
System.out.println(p2.buildHouse(l2, 50));//true
System.out.println(p2.buildHouse(l3, 50));//true
System.out.println(l1);
System.out.println(l2);
System.out.println(l3);
```

Result:

```
true
false
false
false
true
true
Land RED: [H P1: 55]
Land BLUE: [H P2: 50]
Land BLACK: [H P2: 50]
```

Class 5: GameSystem

Fields:

> For this class, do not modify or remove any fields that have been already defined, however you can add other fields if you need.

- LANDS: It is an array type, which represents all lands in the game. The length of lands are in 8-30.

```
private final Land[] LANDS;
```

- players: It is an array type, which represents all players in the game. In our testcases, the length of players are in 2-4.

```
private Player[] players;
```

- gameOver: It is boolean type, the original value is false, if the count of active player is one, the value of gameOver is true.

```
private boolean gameOver;
```

Methods:

> For this class, do not modify or remove any methods that have been already defined, however you can add other methods if you need.

- **Constructor**: Has a two parameter constructor. In this constructor:
    - You should **copy the reference** of parameters to corresponding private fields.
    - Set original value of gameOver is **false**.
    - You can do other original settings if you need.

  ```
  public GameSystem(Player[] players, Land[] lands)
  ```

- **nextPlayer()**: The method returns a Player, which represents the next Player.

  ```
  public Player nextPlayer()
  ```

  The order of player is determined by the index of private fields `players`.

    - When first invoke the method, it will return a player with index 0, then with index 1.
    - When current player is the last player and then invoke the method again, it will return the player with index 0 as long as the player is active.

      For example: the length of `players` is 3, the invoke order is 0->1->2->0->1->2......
    - The invoke order should skip the inactive player。

  For example:

  ```
  Land[] lands = new Land[8];
  for (int i = 0; i < lands.length; i++) {
         lands[i] = new Land(LandColor.values()[i / 2]);
  }
  Player[] players = new Player[3];
  for (int i = 0; i < players.length; i++) {
         players[i] = new Player(i + 1);
  }
  GameSystem gameSystem = new GameSystem(players, lands);
  System.out.println(gameSystem.nextPlayer());
  System.out.println(gameSystem.nextPlayer());
  System.out.println(gameSystem.nextPlayer());
  players[0].setActive(false);
  System.out.println(gameSystem.nextPlayer());
  players[1].setActive(false);
  System.out.println(gameSystem.nextPlayer());
  System.out.println(gameSystem.nextPlayer());
  ```

  Result:

```
Player 1: at 0, has 100.0
Player 2: at 0, has 100.0
Player 3: at 0, has 100.0
Player 2: at 0, has 100.0
Player 3: at 0, has 100.0
Player 3: at 0, has 100.0
```

- **getCurrentPlayer()**: The method returns a Player, which represents the current Player.

  ```java
  public Player getCurrentPlayer()
  ```

  When start the game, the current player is null, and after invoking nextPlayer() method, the current player is not null.

  For example:

  ```java
          Land[] lands = new Land[8];
          for (int i = 0; i < lands.length; i++) {
              lands[i] = new Land(LandColor.values()[i / 2]);
          }
          Player[] players = new Player[3];
          for (int i = 0; i < players.length; i++) {
              players[i] = new Player(i + 1);
          }
          GameSystem gameSystem = new GameSystem(players, lands);
          System.out.println(gameSystem.getCurrentPlayer());
          gameSystem.nextPlayer();
          System.out.println(gameSystem.getCurrentPlayer());
  ```

  Result:

  ```
  null
  Player 1: at 0, has 100.0
  ```

- **isGameOver()**: return the value of private field `gameOver`.

  ```java
  public boolean isGameOver()
  ```

- **dealFailedPlayer()**: The method return a boolean value, which means whether the **current player** is inactive. If the player is inactive, it returns true, otherwise it returns false.

  The method needs to deal with all following actions if a player is inactive:

  - Remove all houses the player has been created in game while other players can build house in the color of the removed house land just now.
  - Check if the count of active player is only 1, change gameOver to true.

```
public boolean dealFailedPlayer()
```

For example:

```
    public static void main(String[] args) {
        Land[] lands = new Land[8];
        for (int i = 0; i < lands.length; i++) {
            lands[i] = new Land(LandColor.values()[i / 2]);
        }
        Player[] players = new Player[3];
        for (int i = 0; i < players.length; i++) {
            players[i] = new Player(i + 1);
        }

        GameSystem gameSystem = new GameSystem(players, lands);
        System.out.println("GameOver:"+gameSystem.isGameOver());
        gameSystem.nextPlayer().buildHouse(lands[0], 80);//RED True
        gameSystem.getCurrentPlayer().buildHouse(lands[1],20);//RED True
        System.out.println(gameSystem.dealFailedPlayer());//false
        gameSystem.getCurrentPlayer().payRent(201);//player 1 failed
        System.out.println(gameSystem.dealFailedPlayer());
        gameSystem.nextPlayer().payRent(201);//player 2 failed
        System.out.println(gameSystem.dealFailedPlayer());
        System.out.println("GameOver:"+gameSystem.isGameOver());//true
        System.out.println(lands[0].getHouse());//null
        System.out.println(lands[1].getHouse());//null
    }
```

Result:

```
GameOver:false
false
true
true
GameOver:true
null
null
```

- **currentPlayersState()**: Return a String[] array, which stores the result of toString() method of each player in original order.

```
public String[] currentPlayersState()
```

- **currentLandsState()**: The returned array stores the result of toString() method of each land whose house is not null in original order.

```
public String[] currentLandsState()
```

For example:

```
public static void main(String[] args) {
        Land[] lands = new Land[8];
        for (int i = 0; i < lands.length; i++) {
            lands[i] = new Land(LandColor.values()[i / 2]);
        }
        Player[] players = new Player[2];
        for (int i = 0; i < players.length; i++) {
            players[i] = new Player(i + 1);
        }
        GameSystem gameSystem = new GameSystem(players, lands);
        players[0].buildHouse(lands[0],50);
        players[1].buildHouse(lands[3],40);
        players[1].buildHouse(lands[4],40);
        for (String s:gameSystem.currentLandsState()) {
            System.out.println(s);
        }
    }
```

Result:

```
Land RED: [H P1: 50]
Land ORANGE: [H P2: 40]
Land YELLOW: [H P2: 40]
```

- **nextTurn()**: This method represents a player do one turn in a game.

```
public void nextTurn(int step, int cost)
```

step : means how many steps the player need to walk. If the player walk to the last land, then he/she will continue to walk from the first land. For example, if the length of `lands` is 10, and the original location of player is 6, the steps and target location are listed below:

steps	target location
1	7
2	8
3	9
4	0
5	1
6	2

cost: means how many money will be cost if the player want to build a house. If the player would not build a house, cost is zero.

In one turn, it includes following actions:

1. Find the **next player**, and the following actions are all initiated by the player that just find.
2. Move to target location and find the target land.
3. If the target land has a house:
    - If the house does not belong to him/her own, the player should pay the rent and the house owner should collect the rent.
    - In this case, if the player doesn't have enough money to pay the rent, the house owner would collect all left money that the player remained.

      > For example,
      >
      > If player1 has 30 money and the house price of player2 is 80, player1 should pay 30 to player2.
      >
      > If player1 has 50 money and the house price of player2 is 80, player1 should pay 40 to player2.

    - Check after paying the rent, whether the player is failed or not. If the player is failed, he/she would release all houses he/she occupied.
4. If the target land has no house:
    - If cost is zero, nothing would happen.
    - If cost is larger then zero, the player can build a house with a price `cost` as long as

      (1) The color of current land isn't occupied by other players.

      (2) The player has enough money.

      If the player can build a house successfully, and he/she would occupied this color of current land.

        - In this case, if current player build a house successfully and other players have already arrived the land, they do not need pay the rent for current player.

          For example:

```
public static void main(String[] args) {
        Land[] lands = new Land[8];
        for (int i = 0; i < lands.length; i++) {
            lands[i] = new Land(LandColor.values()[i / 2]);
        }
        Player[] players = new Player[2];
        for (int i = 0; i < players.length; i++) {
            players[i] = new Player(i + 1);
        }
        GameSystem gameSystem = new GameSystem(players,
lands);
        gameSystem.nextTurn(3,0);//arrived the land 4
        gameSystem.nextTurn(3,40);//arrived the same land 4
and build a house
        for (String s:gameSystem.currentPlayersState()) {
            System.out.println(s);
        }
    }
```

output:

```
Player 1: at 3, has 100.0
Player 2: at 3, has 60.0
```

For other detailed test cases, you can refer Junit Test file.