4. In this question, you will complete methods in classes that can be used to represent a multi-player game. You will be able to implement these methods without knowing the specific game or the players' strategies.

   The `GameState` interface describes the current state of the game. Different implementations of the interface can be used to play different games. For example, the state of a checkers game would include the positions of all the pieces on the board and which player should make the next move.

   The `GameState` interface specifies these methods. The `Player` class will be described in part (a).

```
public interface GameState
{
  /** @return true if the game is in an ending state;
   *          false otherwise
   */
  boolean isGameOver();


  /** Precondition: isGameOver() returns true
   *  @return the player that won the game or null if there was no winner
   */
  Player getWinner();


  /** Precondition: isGameOver() returns false
   *  @return the player who is to make the next move
   */
  Player getCurrentPlayer();


  /** @return a list of valid moves for the current player;
   *          the size of the returned list is 0 if there are no valid moves.
   */
  ArrayList<String> getCurrentMoves();


  /** Updates game state to reflect the effect of the specified move.
   *  @param move a description of the move to be made
   */
  void makeMove(String move);


  /** @return a string representing the current GameState
   */
  String toString();

}
```

   The `makeMove` method makes the move specified, updating the state of the game being played. Its parameter is a `String` that describes the move. The format of the string depends on the game. In tic-tac-toe, for example, the move might be something like `"X-1-1"`, indicating an X is put in the position (1, 1).

(a) The `Player` class provides a method for selecting the next move. By extending this class, different playing strategies can be modeled.

```
public class Player
{
  private String name;    // name of this player

  public Player(String aName)
  {  name = aName;  }

  public String getName()
  {  return name;  }

  /** This implementation chooses the first valid move.
   *  Override this method in subclasses to define players with other strategies.
   *  @param state the current state of the game; its current player is this player.
   *  @return a string representing the move chosen;
   *           "no move" if no valid moves for the current player.
   */
  public String getNextMove(GameState state)
  {  /* implementation not shown */  }
}
```

The method `getNextMove` returns the next move to be made as a string, using the same format as that used by `makeMove` in `GameState`. Depending on how the `getNextMove` method is implemented, a player can exhibit different game-playing strategies.

Write the complete class declaration for a `RandomPlayer` class that is a subclass of `Player`. The class should have a constructor whose `String` parameter is the player's name. It should override the `getNextMove` method to randomly select one of the valid moves in the given game state. If there are no valid moves available for the player, the string `"no move"` should be returned.

GO ON TO THE NEXT PAGE.

(b) The GameDriver class is used to manage the state of the game during game play. The GameDriver class can be written without knowing details about the game being played.

```
public class GameDriver
{
  private GameState state;   // the current state of the game

  public GameDriver(GameState initial)
  {  state = initial;  }


  /** Plays an entire game, as described in the problem description
   */
  public void play()
  {  /* to be implemented in part (b) */  }

  // There may be fields, constructors, and methods that are not shown.
}
```

Write the GameDriver method play. This method should first print the initial state of the game. It should then repeatedly determine the current player and that player's next move, print both the player's name and the chosen move, and make the move. When the game is over, it should stop making moves and print either the name of the winner and the word "wins" or the message "Game ends in a draw" if there is no winner. You may assume that the GameState makeMove method has been implemented so that it will properly handle any move description returned by the Player getNextMove method, including the string "no move".

Complete method play below.

```
  /** Plays an entire game, as described in the problem description
   */
  public void play()
```

STOP

END OF EXAM

AP® COMPUTER SCIENCE A
2007 SCORING GUIDELINES

Question 4: Game Design (Design)

Part A:	RandomPlayer	**4 points**

- **+1/2** `class RandomPlayer extends Player`
- **+1** constructor
  - **+1/2** `public RandomPlayer(String aName)`
  - **+1/2** `super(aName)`
- **+2 1/2** `getNextMove`
  - **+1/2** `state.getCurrentMoves()`
  - **+1** if no moves
    - **+1/2** test if size = 0
    - **+1/2** return "no move" only if 0 moves
  - **+1** if moves
    - **+1/2** select random move index
    - **+1/2** return random move

Part B:	play	**5 points**

- **+1/2** print initial state (OK to print in loop)
- **+3** make repeated moves
  - **+1** repeat until `state.isGameOver()`
  - **+1/2** `state.getCurrentPlayer()`
  - **+1/2** `player.getNextMove(state)`
  - **+1/2** display player and move
  - **+1/2** make move
- **+1 1/2** determine winner
  - **+1/2** `state.getWinner()`
  - **+1/2** display message if draw (if `getWinner` returns null)
  - **+1/2** display message if winner

  lose both if done before game ends

Question 4: Game Design (Design)

PART A:

```
public class RandomPlayer extends Player
{
    public RandomPlayer(String aName)
    {
        super(aName);
    }

    public String getNextMove(GameState state)
    {
        ArrayList<String> possibleMoves = state.getCurrentMoves();
        if (possibleMoves.size() == 0) {
            return "no move";
        }
        else {
            int randomIndex = (int)(Math.random()*possibleMoves.size());
            return possibleMoves.get(randomIndex);
        }
    }
}
```

PART B:

```
public void play()
{
    System.out.println("Initial state:" + state);

    while (!state.isGameOver()) {
        Player currPlayer = state.getCurrentPlayer();
        String currMove = currPlayer.getNextMove(state);
        System.out.println(currPlayer.getName() + ": " + currMove);
        state.makeMove(currMove);
    }

    Player winner = state.getWinner();
    if (winner != null) {
        System.out.println(winner.getName() + " wins");
    }
    else {
        System.out.println("Game ends in a draw");
    }
}
```